2002

The Single Row Routing Problem Revisited: A Solution Based on Genetic Algorithms

Albert Y. Zomaya

Roger Karpin

Stephan Olariu
Old Dominion University

VLSI Design, 2002 Vol. 14 (2), pp. 123–141

The Single Row Routing Problem Revisited: A Solution Based on Genetic Algorithms

ALBERT Y. ZOMAYA[a,*], ROGER KARPIN[b] and STEPHAN OLARIU[c]

[a]*School of Information Technologies, The University of Sydney, NSW 2006, Australia;* [b]*Parallel Computing Research Laboratory, Department of Electrical and Electronic Engineering, The University of Western Australia, Nedlands, Perth, WA 6907, Australia;* [c]*Department of Computer Science, Old Dominion University, Norfolk, VA, USA*

(Received 17 March 1999; Revised 20 April 1999)

With the advent of VLSI technology, circuits with more than one million transistors have been integrated onto a single chip. As the complexity of ICs grows, the time and money spent on designing the circuits become more important. A large, often dominant, part of the cost and time required to design an IC is consumed in the routing operation. The routing of carriers, such as in IC chips and printed circuit boards, is a classical problem in Computer Aided Design. With the complexity inherent in VLSI circuits, high performance routers are necessary. In this paper, a crucial step in the channel routing technique, the single row routing (SRR) problem, is considered. First, we discuss the relevance of SRR in the context of the general routing problem. Secondly, we show that heuristic algorithms are far from solving the general problem. Next, we introduce evolutionary computation, and, in particular, genetic algorithms (GAs) as a justifiable method in solving the SRR problem. Finally, an efficient $O(nk)$ complexity technique based on GAs heuristic is obtained to solve the general SRR problem containing n nodes. Experimental results show that the algorithm is faster and can often generate better results than many of the leading heuristics proposed in the literature.

Keywords: CAD; Genetic algorithms; Heuristics; Single row routing; VLSI

INTRODUCTION

The design and layout of complex multilayer printed circuit boards (MPCBs) and integrated circuits (ICs) is of central importance in electronic systems today. MPCB design and layout involves the following steps: first, placement of the functional modules of the system on the MPCB, and second, interconnection between the modules on the MPCB in a way suitable for the application, subject to various physical and technological constraints.

The complexity of the global routing problem is so large that these two related problems are usually treated separately. The second problem is a classical problem in Computer Aided Design, applicable to all levels of scale. Despite the fact that a large number of CAD packages for layout are available nowadays, the circuit layout problem is far from solved.

Let us first define a *routing region* to be a continuous area between circuit modules that can be used for routing. A *terminal* is a pin on a circuit module. *A signal net* (or simply *net*) is a set of terminals to be interconnected by wires. A *via* is an area where wires on different layers are electrically connected.

The routing task is divided into global routing and detailed routing. Global routing gives an overall analysis on the distribution of nets between modules, generating an intermediate sketch routing for each net. The routing area between circuit modules is divided into a set of routing regions called channels. The global routing result is represented by crossings placed on the interfaces between channels. Note that the global routing does not generate detailed routing, it only specifies a rough path for each net between channels.

The crossing points for the nets between channels do not have fixed locations yet. The exact locations are determined by detailed routing. For a given channel, if all fixed terminal are on one set of parallel edges and non-fixed terminals at the channel ends, then we can use a so-called *channel router* to give a detailed routing of the channel.

The first systematic approach to the general problem of multilayer channel routing was first proposed by So [1].

*Corresponding author. E-mail: zomaya@it.usyd.edu.au

ISSN 1065-514X print/ISSN 1563-5171 online © 2002 Taylor & Francis Ltd
DOI: 10.1080/10655140290010051

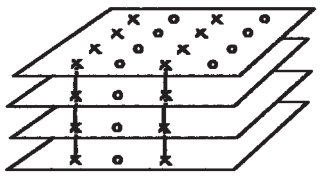


FIGURE 1 An MPCB with pins and vias alternating on each row.

The essence of the layout problem presented by So is to interconnect functional modules, with hundreds or thousands of terminals, by means of printed conductors, layered on a multilayer board.

The large MPCBs consist of pins and vias (feedthroughs), uniformly spaced on a uniform grid. This is shown in Fig. 1. We assume that the placement of modules is predetermined.

So's approach [1] consists of a systematic decomposition of the general multilayer wiring problem into a number of independent single-layer, single-row routing problems. By doing this, he was able to make an estimate on the routability of any given problem. He also developed sufficient conditions for routing, which guarantee routability for the single-row single-layer case. Prior to his work, all techniques have been empirical in nature and lacked the capability of prognostic analysis.

So decomposed the multilayer problem as follows. Consider a backplane with a fixed array of pins and vias as shown in Fig. 2(a). We designate each pin or via according to its location. Thus b_5 represents the pin located at the intersection of the bth row and the 5th column. Suppose that the problem is to route the net list $L = \{N_1, N_2, N_3\}$ where $N_1 = \{a_1, b_5, e_9\}$, $N_2 = \{c_1, c_3, d_5\}$, and $N_3 = \{a_7, c_5, d_7, e_5\}$. This implies that pins a_1, b_5, and e_9 in net N_1 are to be interconnected, as are the pins in N_2 and N_3, respectively.

A possible realization is shown in Fig. 2(b). As shown, So adopts a special strategy that depends on horizontal conductor paths to connect pins and vias which lie on a row, and vertical conductor paths on another layer, as indicated by dotted lines, to connect pins or vias which lie on the same column. This scheme was called unidirectional routing.

This strategy is strategically sound because it allows a systematic study and rules out the necessity of considering other routing strategies. The scheme is also economic and can handle many simple circuits with only two layers. Thus the general multilayer problem in Fig. 2(a) has been reduced to 7 simple single-row single-layer problems.

In general, there are five phases to So's decomposition of the multilayer [1]: via assignment, linear placement of via columns, layering, single row routing (SRR), and via elimination. In this work, we are concerned only with the fourth phase, SRR, which deals with the detailed routing of the single-row single-layer case.

Following the decomposition, there is one SRR problem for every horizontal and every vertical line of points in the original problem. In each single-row routing problem,

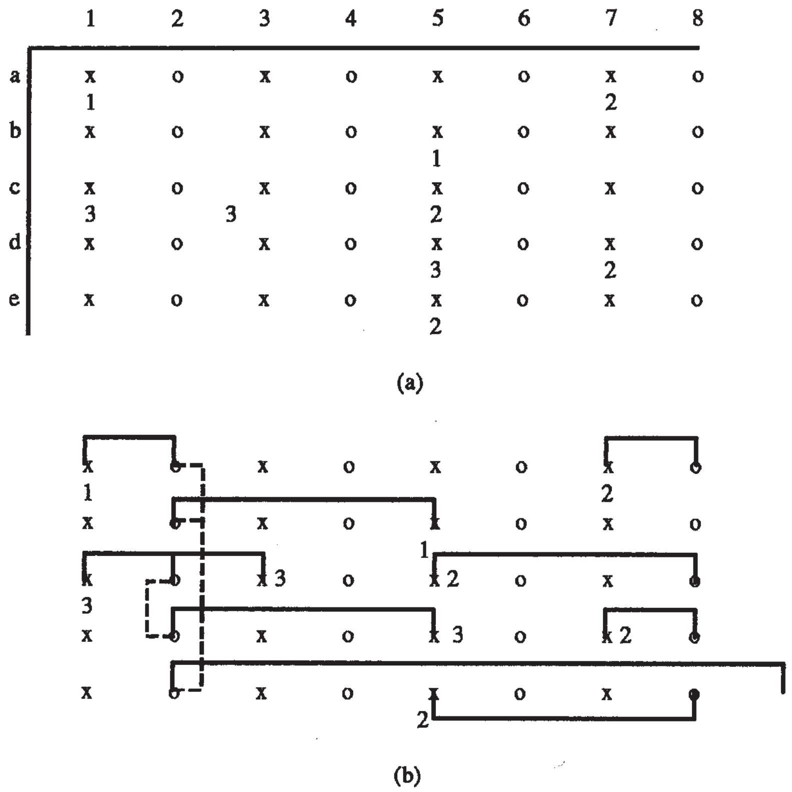
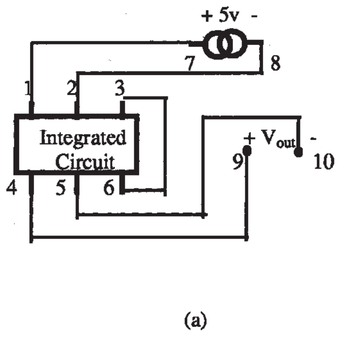


FIGURE 2 Multirow multilayer problem reduced to 7 single-row single-layer problems: (a) 3 nets defined on board with 5 rows and 9 columns, (b) 3 nets connected through vias on board.



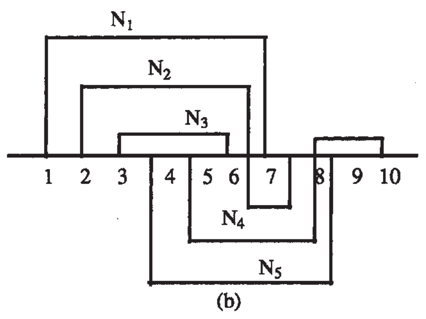


FIGURE 3 (a) Typical circuit, (b) equivalent single row routing.

there is set of evenly spaced nodes and a set of nets. The nets consist of nodes that are to be made electrically equivalent.

The single-row single-layer routing problem can be put in the context of a single layer printed circuit board (Fig. 3(a)). Each module connection can be thought of as a node on a two-dimensional surface. The nodes are interconnected on the PCB by nets of electrical conductor. The nets are routed such that they do not overlap. Now imagine moving all of the nodes into a single row on the two-dimensional surface. The result is a realization of a SRR, as depicted in Fig. 3(b).

So [1] decomposed the multilayer problem as follows. Consider a backplane with a fixed array of pins and vias as shown in Fig. 2(a). We designate each pin or via according to its location. Thus b_5 represents that the pin is located at the intersection of the bth row and the 5th column. Suppose that the problem is to route the net list $L = \{N_1, N_2, N_3\}$ where $N_1 = \{a_1, b_5, e_9\}$, $N_2 = \{c_1, c_3, d_5\}$, and $N_3 = (a_7, c_5, d_7, e_5\}$. This implies that pins a_1, b_5, and e_9 in net N_1 are to be interconnected, as are the pins in N_2 and N_3, respectively.

The SRR problem is very important in the design automation of electronic systems. The problem is known to be computationally intractable [4]. However, an efficient heuristic solution of this problem will have a great impact on other problems of similar nature (e.g. scheduling and networking).

Genetic algorithms (GAs) are a class of computational models particularly suited to solving complex optimization problems efficiently. The goal of this paper is to investigate GAs as a possible approach to solving the SRR problem.

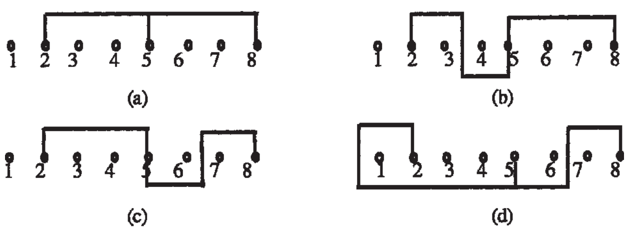


FIGURE 4 (a)–(c) Ways to wire a net, (d) a net with a backward move.

PROBLEM OVERVIEW

In the SRR problem, we are given a set, $V = \{1, 2, \ldots, n\}$, of n nodes that are evenly spaced along a straight line, and a set, $L = \{N_1, N_2, \ldots, N_m\}$, of m nets. Each net, N_i, represents a set of nodes that are to be made electrically equivalent. We say that node j is a touch point of net i if and only if $j \in N_i$. Nets satisfy the following conditions:

$$N_i \cap N_j = \varnothing \quad i \neq j$$

$$\bigcup_{i=1}^{n} N_i = V \quad \{1, 2, \ldots, n\} \tag{1}$$

The nodes may be regarded as module connections or as pins that penetrate all layers of the multilayer board. The straight line on which nodes occur is referred to as the node axis. The wire used to join together the vertices of a net is made up of horizontal and vertical segments. Figure 4 shows some of the possible ways to realize the corresponding routing.

In the development of So [1], arrangements such as the one shown in Fig. 4(d) are not permitted. This arrangement contains a *backward move*; it goes from node 2, around node 1, and then back over node 2. So [1] considers only those wiring schemes where backward moves are not permitted. More formally, if one were to make a vertical cut at any node, the cut can intersect a maximum of only one wire per net.

Finally, horizontal wire segments are run in tracks. Two wires cannot share or overlap a segment of the track. Also, vertical wire segments are not permitted to cross over horizontal wire segments, and vice versa.

A *realization* of a net set L is a wiring scheme that satisfies all of the above requirements. A *realization with backward moves* is a wiring scheme that satisfies all requirements, but allows backward moves. Figure 5shows one SRR realization of the netlist: $n = 10$, $L = \{\{1,7\}, \{2,8\}, \{3,6\}, \{4,9\}, \{5,10\}\}$.

In Fig. 5(a), the area above the node axis is referred to as the *upper street*, and the area below the node axis as the *lower street*. The number of horizontal tracks used in the upper street is called the *upper street congestion* (C_{us}), and the number of horizontal tracks used in the lower street, the *lower street congestion* (C_{ls}). In Fig. 5(a),

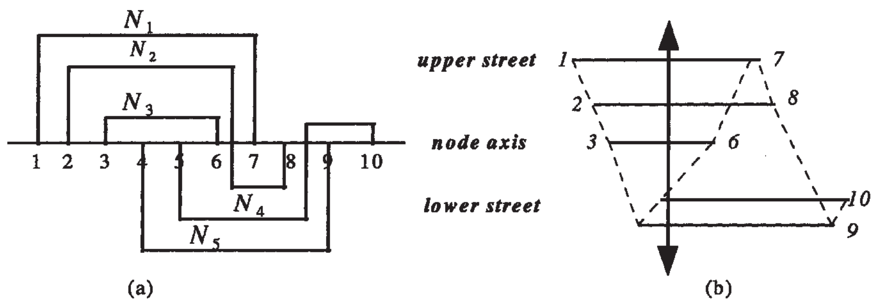


FIGURE 5 (a) Realization of a netlist, (b) interval graph realization of the netlist.

$C_{us} = C_{ls} = 3$. In solving the SRR problem, one tries to obtain a realization which minimizes the street congestion on both streets, or simply to minimize $Q_0 = max\{C_{us}, C_{ls}\}$.

A significant representation of the SRR problem is the interval graph representation. It has been shown in Ref. [2] that each realization of a netlist has a corresponding interval graph representation. The interval graph representation consists of an ordered set of *m* horizontal intervals representing the *m* nets. The node axis is termed the reference line. Each horizontal line corresponds to the interval between the two extreme nodes of a given net. Figure 5(b) shows the interval graph representation of the realization in Fig. 5(a).

In Fig. 5(b), the reference line is a dashed line consisting of continuous line segments connecting the nodes in succession from left to right. If we straighten out the reference line, then the *m* horizontal interval lines are mapped topologically into vertical and horizontal paths. Nets which lie above the reference line are mapped into paths in the upper street, while nets below the reference line are mapped into the lower street.

Let us draw a vertical line at node *i* superimposed onto the interval graph representation, as shown in Fig. 5(b). The number of nets, not including the net to which node *i* belongs, cut by this vertical line is called the *cut number* (c_i) of node *i*. The *upper cut number* (c_{iu}) and the *lower cut number* (c_{il}) of node *i* are defined as the number of nets cut by the vertical line, above and below node *i*, respectively. In Fig. 5(b) above, $c_5 = 4$, $c_{5u} = 3$, $c_{5l} = 1$. Note that, for each node *i*, $c_i = c_{iu} + c_{il}$, and that c_i is fixed for a given instance. That is, c_i will not be affected by the ordering of the nets. However, c_{iu} and c_{il} will certainly be affected by the ordering of the nets. After we straighten out the reference line of an interval graph representation, the number of tracks required above and below the node axis, at node *i*, is equivalent to c_{iu} and c_{il}, respectively. Thus, $C_{us} = \max c_{iu}$ and $C_{ls} = \max c_{il}$.

Nodes on the node axis can be further differentiated by defining a node to be either a *beginning node* (B) if it is at the beginning of a net, i.e. it is the left most node in a net, or an *end node* (E), if it is at the end of a net, i.e. the right most node in a net, or alternatively, a *middle node* (M) if it is a touch point of a net. Thus nets with 2 nodes consist only of a beginning and end node.

SOLVING THE SRR PROBLEM: AN OVERVIEW

By far, most of the research to date for solving the SRR problem is based on heuristics. In this section, we will overview some of the well-known heuristics that have been used to solve this problem.

Existing Solutions for the SRR Problem

The SRR problem has been studied extensively. The problem was first shown to be **NP**-complete by Raghavan and Sahni [3,4]. A brute force approach computing all the possible routings is of order $O(n!)$. This is especially unacceptable in a context where the number of nodes, *n*, is expected to be large.

Some heuristics use the method of trial and error to solve a problem. The problem is broken down into smaller, easier to deal with problems. Then a whole series of trial and error is used to determine the best solution.

A number of researchers have developed necessary and sufficient conditions for a net set to be realizable [2,5]. That is, conditions that are not only adequate for an optimal solution to exist, but also essential. For very limited cases (street widths ≤ 3). Tsukiyama *et al.* [5] developed an $O(mn)$ algorithm to solve the routing problem. A faster algorithm has been developed by Raghavan and Sahni [3]. It has a complexity of $O(k!^*k^*n^*\log k)$ where *k* is the maximum street width. The fastest heuristic found in the literature was developed by Han and Sahni [6,7]. Their algorithm has a complexity of $O(kn)$ for optimally solving the SRR problem on a single layer, however it is restricted to street widths of three or less, i.e. $k \leq 3$.

The drawback of these heuristic algorithms is that they either constrain the problem or produce non-optimal solutions. Heuristics that restrict the street width only deal with a set of "nice" problems that might occur only rarely in reality.

Some Examples of Heuristic Algorithms

In the following we shall examine two heuristic algorithms representing the two main approaches to the SRR problem. These two algorithms are the most efficient ones found that deal with the unrestricted SRR problem. We will use these heuristics to compare our results in "Results" section. The first by Tarng *et al.* [8], is based on cut numbers while the second, by Ting *et al.* [9], is based on necessary and sufficient conditions.

The Algorithm by Tarng et al. [8]

This algorithm is based on the intuitive assumption that nets containing a node with the largest cut number should appear as inner nets on the interval graph representation, while those with the least cut number should appear as outer nets. The algorithm was reported to be of $O(mn)$ complexity, the fastest algorithm found with unrestricted street size.

Before the details of this algorithm can be presented, we need to define the necessary terminology. As previously discussed, the *cut number* (c_i) of net N_i is the maximum cut number of all the nodes that belong to the net N_i. Partition the net list L, into two sub-lists, L' and L'' such that:

$$1.\quad L' \cap L'' = \varnothing \tag{2}$$

$$2.\quad L' \cup L'' = L$$

Now, define the *internal cut number* (ic_j) of the net N_j with respect to L' as the cut number of N_j in L'. The *residual cut number* (rc_j) of net N_j with respect to L' is defined as the cut number of N_j in L''.

In this algorithm, we first group all nets, in the netlist L, into several "classes," $L_0, L_1, \ldots, L_k$. Let c_M denote the maximum cut number of all the nets, i.e.

$$c_M = \max(c_1, c_2, \ldots, c_n) \tag{3}$$

Then, a net N_j with cut number i is assigned to $L_{c_M} - i$. And so, all of the nets are grouped according to their cut numbers, with nets of cut number $c_j = c_M$ in class L_0, up to the class L_k containing nets with lowest cut numbers. After all nets are grouped into classes, the internal and residual cut numbers for all nets in each class L_i with respect to L_i', (the union of all nets in that class), is calculated.

The next step is to sort all nets according to their class, internal and residual cut numbers. A net with smaller class index comes before a net with larger class index. Nets with the same class are arranged according to descending internal cut numbers. If two nets belong to the same class and have the same internal cut number, then the one with larger residual cut number precedes the one with smaller residual cut number. If two nets belong to the same class and have the same internal and residual cut numbers, then the ordering can be arbitrary. It was reported in Ref. [8] that the algorithm always produced optimal solutions for various examples. In the following, we present an example where the algorithm does not generate an optimal solution.

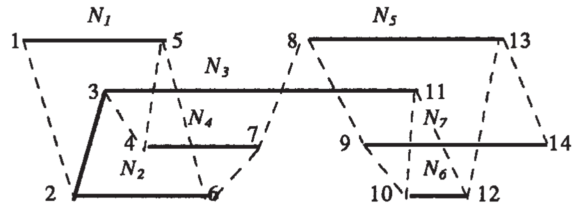


FIGURE 6 Interval graph representation of one solution ($Q_0 = 3$).

Example of Tarng et al. Algorithm [8]

Let $L = \{N_1, N_2, N_3, N_4, N_5, N_6, N_7\}$ where $N_1 = \{1,5\}$, $N_2 = \{2,6\}$, $N_3 = \{3,11\}$, $N_4 = \{4,7\}$, $N_5 = \{8,13\}$, $N_6 = \{10,12\}$, $N_7 = \{9,14\}$. The cut number (c_j), internal cut number (ic_j), and residual cut number (rc_j) of each net N_j are given in Table I.

The nets are first partitioned into classes according to their cut numbers. Thus, $L_0 = \{N_1, N_3, N_4, N_6\}$, $L_1 = \{N_2, N_7\}$, and $L_2 = \{N_5\}$. Then the nets are further sorted based on their internal and residual cut numbers. Thus, one possible order of all nets is N_1, N_4, N_3, N_6, N_2, N_7, N_5. The corresponding interval graph representation obtained is shown in Fig. 6, and the street congestion is, $Q_0 = 3$.

Let c_m and c_M denote the minimum and maximum cut number of all nets, respectively. Let $\lceil y \rceil$ denote the smallest integer greater than or equal to y. It has been shown in Ref. [2] that for each realization,

$$Q_0 \geq \max\{c_m, \lceil c_M/2 \rceil\} \tag{4}$$

For the above example a realization with $Q_0 = \lceil c_M/2 \rceil = 2$ is certainly optimal. A realization with $Q_0 = 2$ is shown in Fig. 7. Thus, it can be seen that the algorithm proposed in Ref. [8] does not produce optimal results even for simple netlists.

The Algorithm by Ting et al. [9]

The heuristic algorithm proposed by Ting *et al.* [9] represents the second major paradigm of the heuristic approach to solving the SRR problem. Ting's algorithm is based on necessary and sufficient conditions for finding the optimal solution. It can potentially take exponential time, i.e. $O(e^n)$.

Before we describe the necessary and sufficient condition used in Ref. [9], we need to present a number of definitions.

TABLE I Cut numbers, internal cut numbers, and residual cut numbers of each net

Net	Cut number	Internal cut number	Residual cut number
1	3	2	1
2	2	2	0
3	3	1	2
4	3	2	1
5	1	1	0
6	3	1	2
7	2	1	1

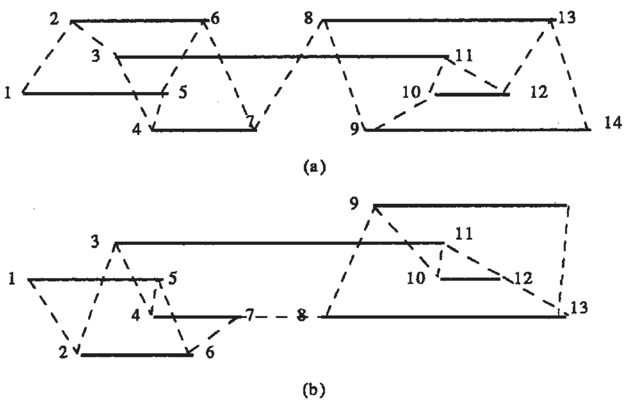


FIGURE 7 The interval graph representation with (a) $Q_0 = 3$, (b) $Q_0 = 2$.

- A net N covers an interval $[c,d]$, if its beginning and end nodes, v_b and v_e, respectively, surround the interval $[c,d]$, i.e. $v_b <= c < d <= v_e$. For the case of $c = d$, we say that the net N covers the node c or d.
- With respect to the interval graph representation, a net N covers a node c from above if net N covers node c, and the portion of net N at node c is in the upper street. Similarly, a net N covers node c from below if the portion of net N at node c is in the lower street.
- An interval $[c,d]$ is of the type $I(k)$ if all nodes in the interval have their cut numbers no less than k and the preceding node (i.e. node $c - 1$) and the succeeding node (i.e. node $d + 1$) of the interval have their cut numbers equal to $k - 1$.
- The density of a unit interval $[a, a + 1]$ is the number of nets covering the interval. We denote the maximum density of all unit intervals on the node axis as ρ. For a given set of nets, the minimum street congestion for all possible realizations is $Q_0 \geq \lceil \rho/2 \rceil$.

It is interesting to note that Liu *et al.* [10] obtained a tighter lower bound on the street congestion based on the density of unit intervals. Let $\lfloor y \rfloor$ denote the largest integer less than or equal to y. Then the street congestion Q_0 for the optimal realization satisfies

$$Q_0 \geq \max\{c_{\mathrm{m}}, \lfloor c_{\mathrm{M}}/2 \rfloor + 1\} \tag{5}$$

The necessary and sufficient condition used in Ref. [9] can be stated as follows: there exists an optimal realization with street congestion Q_0 iff for each unit interval with density $I > Q_0$, there is at least $2(I - Q_0)$ nets covering the interval and each of them has cut number less than I.

Initialize $x = \lceil \rho/2 \rceil$, and sort all unit intervals with density greater than x based on their density, (smaller first), and sequence on the node axis. Then, according to the sorted order, each unit interval with density greater than x is checked to see if the sufficient condition is satisfied. If the condition is not satisfied x is incremented by one and the search is continued from the previously failing unit interval.

If the condition is met then some unassigned net which covers the interval and has cut numbers less than I are assigned to the outermost position of the upper and lower streets, such that there are at least $I - x$ nets covering the interval. Once all intervals have been checked, the remaining unassigned nets are assigned to the middle of the interval graph representation. Then, for the realization, $Q_0 = x$. Unfortunately, this algorithm does not always produce optimal results, as shown in the example below.

Example of Ting et al. Algorithm [9]

Using the previous example in the second section of "Some examples of heuristic algorithms" with $L = \{N_1, N_2, N_3, N_4, N_5, N_6, N_7\}$ where $N_1 = \{1, 5\}$, $N_2 = \{2, 6\}$, $N_3 = \{3, 11\}$, $N_4 = \{4, 7\}$, $N_5 = \{8, 13\}$, $N_6 = \{10, 12\}$, $N_7 = \{9, 14\}$ we find that $\rho = 4$, thus we initialize $x = \lceil \rho/2 \rceil = 2$.

According to the algorithm, we only have to examine unit intervals with density greater than 2. All of the remaining unit intervals are sorted into the following order: [3,4], [5,6], [9,10], [11,12], [4,5], [10,11]. We begin by examining the interval [3,4] with density $I = 3$, to see if the sufficient condition holds for this interval, i.e. for unit interval with density of $(I = 3) > (Q_0 = 2)$ there is at least $2(I - Q_0) = 2$ assigned nets covering the interval. The condition cannot be met, therefore, x is incremented to 3. One possible realization obtained by this algorithm is shown in Fig. 7(a). However, an optimal realization only requires $Q_0 = 2$ as shown in Fig. 7(b).

GENETIC ALGORITHMS

A genetic algorithm is a search algorithm which is based on the principles of evolution and natural genetics. GAs combine the exploitation of past results with the exploration of new areas of the search space. By using *survival of the fittest* techniques combined with a structured yet randomized information exchange, a GA can mimic some of the innovative flair of human search. A generation is a collection of artificial creatures (strings). In every new generation, a set of strings is created using information from the previous ones. Occasionally, a new part is tried for good measure. GAs are randomized, but they are not simple random walks. They efficiently exploit historical information to speculate on new search points with expected improvement [11].

The central theme of research on GAs has been *robustness*. The balance between efficiency and efficacy necessary for survival in many different environments. The implications of robustness for artificial systems are manifold. If artificial systems can be made more robust, costly redesigns can be reduced or eliminated. If higher levels of adaptation can be achieved, existing systems can perform their functions longer and better. Features for self-repair, self-guidance, and reproduction are the rule in biological systems, whereas they barely exist in the most sophisticated artificial systems [12–15].

In order for GAs to surpass other techniques in terms of robustness, they must differ in some fundamental ways. GAs are different from more normal optimization and search procedures in five ways: (1) working with a coding of the parameter set, not the parameters themselves; (2) searching from a population of points, not a single point; (3) using payoff (objective function) information, not derivatives or other auxiliary knowledge; (4) using probabilistic transition rules, not deterministic rules; and (5) coding.

The majority of optimization methods move from a single point in the decision space to the next using some transition rule to determine the next point. This point-to-point method is dangerous as it can locate false peaks in multimodal (many-peaked) search spaces. By contrast, GAs work from a database of points simultaneously (a population of strings), climbing many peaks in parallel. The probability of finding a false peak is reduced compared to methods that go point to point.

Many search techniques require auxiliary information in order to work properly. For example, gradient techniques need derivatives in order to be able to climb the current peak, and other local search procedures like the greedy techniques of combinational optimization require access to most if not all tabular parameters. GAs have no need for all this auxiliary information, they are blind. To perform an effective search for better and better structures, they only require payoff values (objective function values) associated with individual strings. This characteristic makes a GA a more canonical method than many search schemes. Different search problems have vastly different forms of auxiliary information. By not using this auxiliary information, a broadly based scheme can be developed. Of course, this does not mean that when information is available one should not use it.

The mechanics of a simple GA are surprisingly simple, involving nothing more complex than copying strings and swapping partial strings. Simplicity of operation and power of effect are two main attractions of the GA approach. The effectiveness of the GA depends upon an appropriate mix of *exploration* and *exploitation*. Three operators to achieve this are: *selection*, *crossover*, and *mutation*.

Selection according to fitness is the source of exploitation. The mutation and crossover operators are the sources of exploration. In order to explore, they must disrupt some of the strings on which they operate. The tradeoff of exploration and exploitation is clearest with mutation. As the mutation rate is increases, mutation becomes more disruptive until the exploitative effects of selection are completely overwhelmed. More information is provided on these operators in the next section.

The Workings of a GA

A GA starts with a pool of feasible solutions (population) and a set of biologically inspired operators defined over the population itself. At each iteration, a new population of solutions is created by breeding and mutation, with the fitter solutions being more likely to procreate. According to evolutionary theories, only the most suited elements in a population are likely to survive and generate offspring, transmitting their biological inheritance to the next generation. GAs operate through a simple cycle of stages: creation of a population a strings, evaluation of each string, selection of the best strings, and reproduction to create a new population.

Individuals are encoded as strings, termed chromosomes, composed over an alphabet. The chromosome values, termed genotypes, are uniquely mapped onto the decision variable, phenotypic, domain. The most common representation for GAs is the binary alphabet {0,1}. Other representations include ternary, integer and real valued.

Variables are mapped onto the chromosome. When the chromosome is decoded into its phenotypic values, meaning specific to the problem can be gained.

Once the chromosome has been decoded, it is possible to evaluate the performance, or fitness, of individuals in a population. An objective function is used to characterize an individual's performance to the problem. This is analogous to an individual's ability to survive in the natural world. Thus, the objective function gives the basis for selection of pairs of individuals that will be mated together during reproduction. During selection, each individual is assigned a fitness value given by the objective function. Then pairs are selected for matting. Individual selection is biased to fitter individuals, giving them a proportionally higher chance of being selected.

Reproduction involves two types of genetic manipulation, namely crossover and mutation. The simplest crossover operator is single point, where genetic information is swapped after a random position, producing two new offspring. Mutation is another genetic operator that is applied to all new chromosomes with a set probability. In the binary string representation, mutation will cause a random bit to change its state, 0 to 1 or vice versa. Mutation can be considered as a background operator that ensues the probability of finding the optimal solution is never zero. Mutation tends to inhibit the possibility of converging to a local, rather than the global optimum.

After reproduction, the cycle is repeated. New individuals are decoded and the objective function evaluated to give their fitness values. Individuals are selected for mating according to fitness, and so the process continues. The average performance of individuals in a population is expected to increase as good individuals are preserved and bred, while less fit members die out. The GA is terminated under a given criteria, for example, a certain number of generations have been completed, a level of fitness has been obtained or a point in the search space has been reached.

There are several parameters to fine tune in a GA, such as population size and mutation frequency. These parameters can be chosen with experience, or though experiments.

Modifying Simple Genetic Algorithms

The basic type of GAs, known as the simple GA (SGA), uses a population of binary strings, single point crossover and proportional selection [11]. Many other modifications to the SGA have been proposed, some of these are used in this work.

Population

Typically a SGA uses of a population of between 30 and 100 individual solutions, although a variant called the *micro GA* uses a very small population, ~ 10 individuals, in order to speed computation time.

Initialization and Realization

The first step in the SGA is to create an initial population. Usually a random number generator is used to uniformly distribute numbers in the desired range. For instance, a binary population of N_{ind} individuals whose chromosomes are L_{ind} bits long would require, $N_{\text{ind}} \times L_{\text{ind}}$ random numbers uniformly distributed over the set $\{0,1\}$ to be generated. A variation to this is the *extended random initialization* where the GA is seeded with individuals known to be in the vicinity of the global minimum.

Fitness and Objective Functions

The objective function provides the mechanism for evaluating each chromosome in the problem domain. In the case of a minimization problem, the most fit individuals would have the lowest numerical value for their objective function. The fitness function normalizes the objective function value, transforming it into a relative measure of fitness in a convenient range, 0–1, i.e.

$$F(x) = g(f(x)) \tag{6}$$

Here, f is the objective function, g transforms the value of the objective function to a non-negative number and F is the resulting relative fitness. The normalized fitness value is then used by the selection mechanism.

Selection

Selection models the "survival of the fittest" mechanism. Fitter solutions survive while weaker ones perish. In the SGA, a fitter string is more likely to receive a higher number of offspring, increasing its chances of survival.

In the *proportionate selection scheme*, where a string with fitness value F_i is allocated a relative fitness of $F_i/\underline{F}$, where $\underline{F}$ is the average fitness of the population. The SGA uses the *roulette wheel* style of selection to implement proportional selection. Each string is allocated a sector (slot) of a roulette wheel with the angle subtended by the sector at the center of the wheel equal to $2\pi F_i/\underline{F}$. A string is allocated an offspring if a randomly generated number in the range 0–2π falls in the sector corresponding to the string. The algorithm selects strings until the next generation is completely generated.

The basic roulette wheel selection method is called stochastic sampling with replacement (SSWR). With this method, the segment size and corresponding selection probability remain the same throughout selection. It is also possible for the final number of offspring to vary significantly from that expected. However, for a large population, the actual number of offspring approaches that expected.

Stochastic sampling with partial replacement (SSWPR) extends upon SSWR by reducing the sector of an individual if it is chosen. Another extension is *remainder SSWR* (RSSR). Here individuals are selected according to the integer part of their expected number of offspring, with the fractional part decided probabilistically, either SSWR or SSWPR. Other types of selection techniques are SSWPR and *Stochastic universal sampling* (SUS).

Crossover

Crossover produces new individuals that have some parts of both parents' genetic material. The simplest form of crossover is single-point crossover, which was described previously. Typically the SGA uses a crossover rate of between 0.5 and 1.0.

Multipoint crossover uses m randomly chosen crossover positions. Bits between successive crossover points are exchanged producing two new offspring. In this case, parts of the chromosome that contribute most to the fitness of an individual may not necessarily be contained in adjacent substrings. The disruptive nature can also encourage exploration of the search space, rather than favor convergence to locally fit individuals early on, making the search more robust. Other methods of crossover are *Uniform* and *Shuffle* crossovers. Real-valued genes can also use *line recombination* or *intermediate recombination* schemes.

Mutation

As stated earlier, strings are subject to mutation. Mutation is applied uniformly to the entire current generation of strings. In SGAs, mutation is randomly applied with a low probability, typically in the range of 0.1–1.0%. In the SGAs, mutation is a background operator, ensuring that the probability of finding the optimal solution is never zero. Mutation also acts as a safety net to recover good genetic material that may be lost through selection and crossover. Variations on mutation include *biasing* towards less fit individuals, increasing the exploration without losing information from fitter individuals, or *changing* the mutation rate, decreasing it with population convergence or increasing it with stagnation. With non-binary

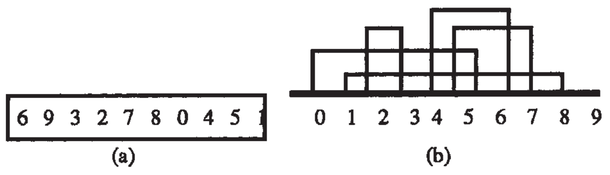


FIGURE 8 (a) A node net list with $m = 5$, (b) Net diagram.

representations, mutation is achieved by randomly altering the gene values within some allowed range.

Termination

The GA is a stochastic search method where the average performance of individuals in a population are expected to increase as good individuals are preserved and bred, while less fit members die out. However, because a population may remain static for a number of generations before a superior individual is found, using single termination criteria is problematic. Typically, a GA is terminated after certain number of generations, or if a level of fitness has been obtained or a point in the search space has been reached.

THE PROPOSED METHODS AND IMPLEMENTATION DETAILS

This section details the implementation of a GA and then its application to the SRR problem. Then, the implementation of Tarng *et al.*'s [8] heuristic algorithm will also be investigated.

Single Row Netlist

In this section, a routing environment is setup and then GA-based techniques are developed to solve the SRR problem.

The first step in creating a routing environment is to generate a representation of a netlist, $L = \{N_1, N_2, \ldots, N_m\}$, of m nets and n nodes. The nets contain nodes that are to be made electrically equivalent. A netlist needs to be generated at random or taken from an example circuit. Initially, we consider 2 node nets, with $n = 2 \times m$, but as agreed in Ref. [4] this restriction is not essential. Thus the netlist needs the following characteristics: readily understood format, quick generation of examples, possibility of expanding beyond 2 node nets, simple assimilation into an input program.

The simple "nodelist" format shown in Fig. 8has all of these features and takes up a minimal amount of memory. Each number represents a node on the node axis, with the value of the number giving the node to be made electrically equivalent with it. Thus, we can see that node zero is included in the net with node 6, and vice versa. There are many features in this representation that can be exploited by routing algorithms.

The next step is to create a random nodelist generating algorithm that will generate examples for the routing algorithm to be developed later. The nodelist generating algorithm begins with an empty array of length m. Nodes are considered from left (beginning) to right (end). If a node is already part of a net, then the next node is considered, as shown in Fig. 9.

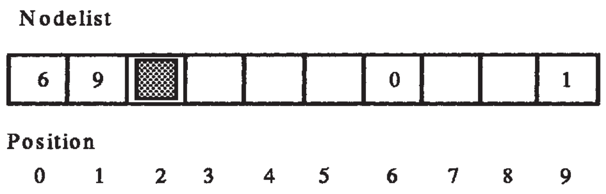


FIGURE 9 A netlist array. Node (2) is the next node to be considered, all possibilities lie to the right in the array.

An unpaired node is then chosen at random, and a net pair is formed. All unpaired nets are guaranteed to occur to the right of the node being considered. If a node is chosen that is part of a net, the algorithm tries again. At first glance, this may continue indefinitely, however, it was found after a number of trials that the total number of nodes considered was in direct proportion to the number of nodes. In fact, as n increased the number of nodes considered approached $1.5 \times n$. This is quite acceptable.

Other methods would involve a look-up table where all unrouted nodes are put in a table for selection. However, these methods would drive the complexity up from $O(n)$ to $O(n^2)$.

The Single Row Router

Once an example nodelist has been generated and stored in a file, it is ready to be routed. The algorithm Genetic Single Row Router (GSRR) reads a nodelist from the file specified in the command line, finds the cut numbers of each node and then executes the genetic routing algorithm to find an optimal routing.

Cut Numbers

Once the netlist has been read into the data structure, the next step is to determine the cut numbers (c_i) of every node, $i \in [0, n-1]$. The cut numbers can be simply obtained from the netlist format.

Nodes in a two-node net are either beginning nodes (B) or end nodes (E). Beginning nodes are always paired with nodes to the right, while end nodes are always paired with nodes to the left. As a result one simply has to examine a nodelist:

- if nodelist[i] $> i$, then the node is a beginning node, (B).
- if nodelist[i] $< i$, then the node is an end node, (E).
- if nodelist[i] $= i$, then the node points to itself, not possible in a two node net.

The cut number of a net can be thought of as the number of active nets over a node, where a node is active if the

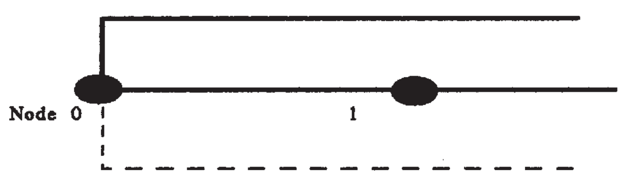


FIGURE 10 Starting the routing process.

beginning node has been reached, but the end node is yet to be encountered. Thus, the cut number of a node is the sum of all previous (B) type nodes minus the sum of all previous (E) type nodes. The code for the cut number function is given below.

From the cut number, the theoretical lower bound on the street width at node i, $C_{i,\min}$, can be determined. The minimum possible value of $C_i = \max(c_{i\mathrm{u}}, c_{il})$ occurs when $c_{i\mathrm{u}} = c_{il}$, since from Eq. (2), $c_i = c_{i\mathrm{u}} + c_{i\mathrm{l}}$ where c_i is a fixed number, thus:

$$C_{i,\min} = \lceil c_i/2 \rceil \tag{7}$$

This value is determined for every node in the nodelist. The theoretical upper bound on the street width occurs when $c_{i\mathrm{u}} = 0$, or $c_{il} = 0$, thus

$$c_{i,\max} = c_{\mathrm{i}} \tag{8}$$

Applying the GA to the SRR

Once all relevant preliminaries have been completed, the genetic routing algorithm is used to find the optimal routing. The GA used is based on the one presented in “Single row netlist” section with two major differences: the objective function, $f(x)$, and the fitness function, $g(x)$.

We will first consider the crucial objective function, which transforms a chromosome to a routing and then evaluates attributes of the routing to produce a value. The success or failure of the GSRR hinges on the objective function.

The Routing Objective Function

At the outset of each generation of the genetic routing algorithm the fitness of each string has to be determined. Each chromosome, represented by a string, has a value and an associated fitness. The objective function, `DecodeRouting (n, i)`, takes string i, and netlist n, and produces the required value.

Routing Decisions

Genetic algorithms can be used in a problem by first considering what choices have to be made in order to generate an arbitrary solution. Let us first consider the start of an arbitrary SRR to see what decisions need to be made (see Fig. 10). We see that only 2 choices are possible, to route the first net above or below the reference line. This is a binary decision that can be implemented as a binary bit on the chromosome. Next we consider the more general case.

In Fig. 11we have a net, N_i, containing two nodes, $N_i = (\alpha, \beta)$, where α is to the left of β, thus, α is a start node and β is an end node.

We see that around end nodes no routing decisions need to be made, since to form a solution without backtracking, net N_i must always be routed to β. This is shown in Fig. 11(a). Before the start node the number of pseudo points, $p_{\alpha,}$ needs to be decided. The set of pseudo points, $Q_\alpha = \{q_1, q_2, \ldots, q_p\}$, are the points of intersection of wiring paths and the node axis, Fig. 11(b).

The number of pseudo points is an integer decision about the routing at a start node. This integer decision is a gene that can be implemented as K binary bits on the chromosome. Consequently, the range of possible pseudo points is $[0, 2^K - 1]$. The larger the value of N we choose, the larger the search space. However, much search time can be wasted for values out of the possible range.

Another type of routing decision was thought to exist between two adjacent start nodes, since a binary routing decision exists. N_α can only be routed above or below the following node, as shown in Fig. 11(c). However, this is just a special case of the previous routing decision, where other nets are not permitted to cross the node axis between two adjacent start nodes.

The first routing decoder implemented only allowed a single binary decision between two adjacent start nodes. The second allowed a binary decision about the number of pseudo points before all start nodes, while the final version allowed the number of pseudo points before all start nodes to be an integer. The GSRR program proposed here allows any of these schemes, by specifying the value of K as a

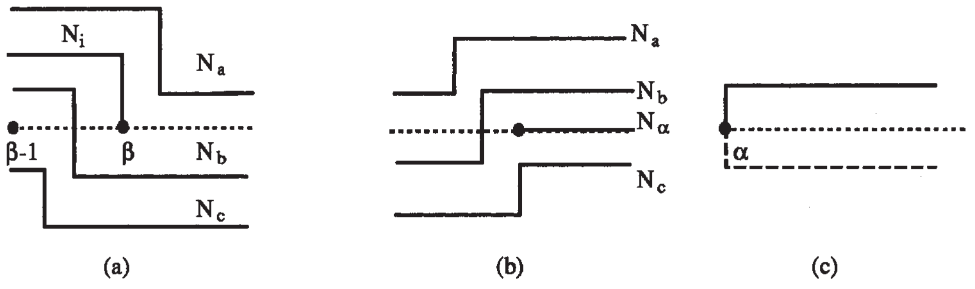


FIGURE 11 (a) Net routing to end nodes, (b) Net routing before and, (c) after start nodes.

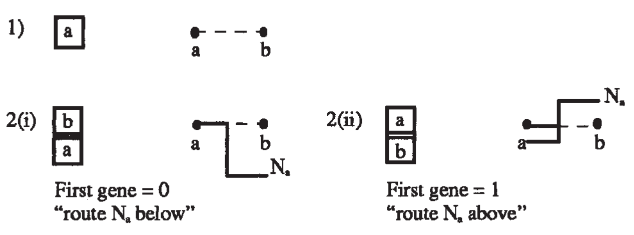


FIGURE 12 The first routing decision, (1) initial 2(i) or 2(ii) routed.

command line argument, $K = 0$ commands the router to use the first scheme, $K = 1$, the second, and $K = x, x < 1$ for the third.

Chromosome Length

An additional step required by a variable encoding scheme to determine the number of bits needed in a chromosome, termed the chromosome length. One chromosome contains all the information to produce a routing, with bit positions fixed for any particular nodelist. The function `ChromLength (n)`, shown below, takes a routing n, and returns the chromosome length needed for the encoding. It simply counts the number of start nodes minus one, and multiplies the value by K. In the case of $K = 0$, only adjacent start nodes are counted.

```
int ChromLength (n)

nodelist *n;

{

int i, length = 0;

for (i = 1; i < n- > size; i++)

if(n- > node[i] > i) /* start node */

if ((n- > node[i-1] > i) &&
        (maxpseudo = = 0))

  length + = 1; /* followed by start */

else

  length + = maxpseudo;

          return (length);

          } /* End of ChromLength ( ).
          */
```

When the `initialize_population ( )` function is called at the beginning of the genetic routing algorithm, all strings are initialized to `ChromLength (n)` size.

Implementation of the Routing Objective Function

To implement the objective routing function, we need a data structure to hold the order of the nets at each node. Four procedures are also required to manipulate it. The router starts at the first node, always a start node, adding its net number to the data structure. The router continues to the right, using the chromosome to make routing decisions while manipulating the data structure accordingly. Taking the second node,

- if it is an end node then the net must be removed from the data structure,
- if it is a start node, then it must be added to the data structure in the order specified by the first gene on the chromosome.

This is shown below in Fig. 12. To complete the picture, only the position of the reference line needs to be known.

Over the entire routing the data structure is added to (in the case of start nodes), and removed from (in the case of

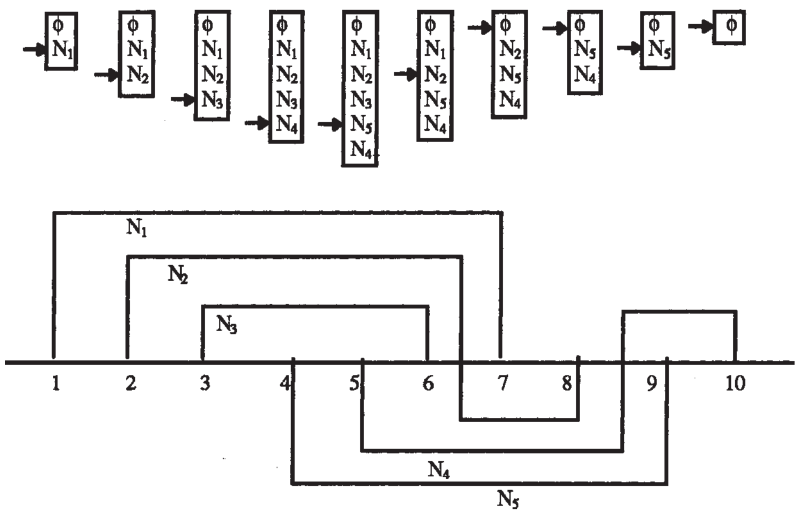


FIGURE 13 A routing example and the associated data structure.

end nodes). An example routing is shown in Fig. 13, with the state of the data structure at each node shown. There are a few items of interest relating to Fig. 13. The routing diagram itself can be derived from the data structure and vice versa. Therefore, a printout of the data structure at each stage will give a picture of the routing for a given chromosome. A convention is needed concerning the position of the reference line, since a net is never on an entire interval of the reference line. The convention applied is that the net on or directly above the reference line is used to show (flag) its position. Where all nets are below the reference line, the flag points to the null net which always sits above all others, e.g. nodes 7 and 8 in Fig. 13. The null net can simply be the first element of an array, list, or tree. Four routing functions are needed to fully manipulate the data structure, by: adding a start node, searching for an end node, removing an end node, and finding upper and lower street widths.

Let us consider the general case of adding a start node z, to the netlist as shown below in Fig. 14.

Obviously, the reference line will always be at node z after inclusion. The routing decision is made before the node is added. If net c is to be routed above node z, the reference line flag is moved up at least one position. The number of positions depends on the number of pseudo points, p_α, decided by the chromosome.

At an end node, no routing decisions need to be made. The data structure is searched for net α, and the reference line flag set to it. Then net α needs to be removed from the nodelist and the flag set to the next higher net.

The final function required is to determine the street widths from the data structure. The street width is determined just before each start node and after each end node. The street widths are used to determine the value of the objective function, which results in a fitness value.

Possible Choice of Data Structures

The routing data structure and the resulting manipulation functions are of prime importance, since they determine the order of complexity of the entire GSRR. An implementation that can minimize the amount of computation and storage space, is required.

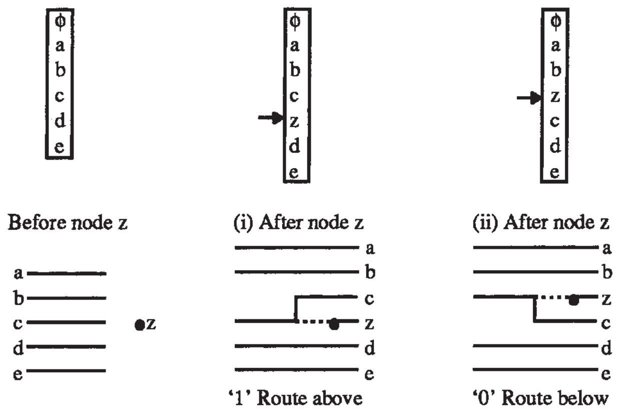


FIGURE 14 Adding a start node to the data structure.

Table II shows all of the data structures considered, their features and complexities.

Two data structures were implemented in the router, arrays for ease of implementation and doubly linked lists for lowest computational cost.

List Implementation

Of all the data structures considered, doubly linked lists are theoretically the least computationally intensive. Lists are built out of "CUTNODES" used to represent nets in the cut. The order of the nets gives information about the routing. A CUTNODE consists of two pointers and an integer to hold the net number, as shown below. The pointers form a doubly linked list, one pointer to higher and one to a lower CUTNODE in the `list.possible` data structures.

The reference line flag is implemented as a pointer to the relevant CUTNODE. The null net is always at head of the list, and is implemented by a CUTNODE with net number set to -1. Several functions were implemented to cater for the manipulation of the doubly linked list.

1. `AddToList (CUTNODE * cut, int net)` used to start a new netlist with net number, *net*. A new CUTNODE is created and placed above *cut*. A pointer is returned to the new CUTNODE.

TABLE II Possible data structures

Data structure	Functional order	Suitability
1 Array	$O(k)$ $O(k)$ $O(k)$ $O(1)$	Very suitable, easy to implement
2 Arrays	$O(k)$ $O(k)$ $O(k)$ $O(1)$	Suitable, one lower street, and one upper street
Stacks	NA (not applicable)	FIFO, unsuitable
Queues	NA	LIFO, unsuitable
Lists	$O(1)$ $O(1)$ $O(1)$ $O(k)$	Very suitable, quick search and manipulation
Binary trees	$O(k)$ $O(1)$ $O(k)$ $O(k)$	Suitable, slow searching
Tables	$O(k)$ $O(k)$ $O(1)$ $O(k)$	Suitable, slow manipulation
N-ary trees	NA	Unsuitable, no obvious structure
Symbolic table	NA	Unsuitable, no inherent key, no gain over table
AVL trees	$O(\log k)$ $O(k)$ $O(k)$ $O(k)$	Balancing destroys inherent street ordering
Hashing	NA	Not a sparse table, key not applicable
Priority queues	NA	Unsuitable, no inherent priority
Sets	NA	Unsuitable for adding or removing

2. `InsertToList (CUTNODE *cut, int net)` inserts a new CUTNODE below that pointed to by *cut*, and returns a pointer to the new CUTNODE.
3. `FindNode (CUTNODE *thisnet, CUTNODE *cut)` searches for *thisnet* in the *cut* list, and returns a pointer to it. The position of *thisnet* in the list, replaces the net number in *thisnet*. The value is needed to determine street widths.
4. `RemoveFromList (CUTNODE *thisnet, CUTNODE *cut)` removes the CUTNODE passed to it and returns a pointer to the CUTNODE immediately above it. Attempting to remove a null node or a node in the list without a higher node, results in an error.

`DecodeRouting` was also implemented as an array. The comparison between the two schemes is given in the next few pages.

Objective Function

In order to determine the fitness of a routing, the value of the objective function needs to be evaluated. At each node, the street widths are calculated from the routing data structure. This, together with cut number information, gives all that is required to evaluate the fitness of a routing. The upper street congestion, C_{us}, is simply the maximum of the upper street widths, $c_{i\mathrm{u}}$:

$$C_{\mathrm{us}} = \max(c_{i\mathrm{u}}) \tag{9}$$

and similarly for the lower street:

$$C_{\mathrm{ls}} = \max(c_{i\mathrm{l}}) \tag{10}$$

An optimal realization is one which minimizes the street congestion in both streets. Thus, the objective of SRR is to minimize:

$$Q_0 = \max\{C_{\mathrm{us}}, C_{\mathrm{ls}}\} \tag{11}$$

The selection process biases fitter strings with higher fitness values. So, the minimization of the street congestion must first be converted to a maximization. First, we define the *maxcut* as maxcut $= \max(c_{\mathrm{i}})$. Then highest value of Q_0 possible is given by the maximum cut number, so the objective of SRR can be restated as to maximize maxcut $- Q_0 = \max(c_i) - \max\{C_{\mathrm{us}}, C_{\mathrm{ls}}\}$. The first objective function value tried was the one given in the previous equation. The results for the different objective functions are given in "Results" section, as well as the reasoning behind their formulation.

The Fitness Function, $g(x)$

Every solution string has a value and an associated fitness. The fitness function, $g(x)$ transforms the objective value into a non-negative relative fitness, F. The normalized fitness value is then used by the selection mechanism to bias reproduction to fitter routings. The higher the relative fitness, the larger the chance that a string has of being selected to pass its genetic material to the next generation.

As the breeding process continues, a record needs to kept of the best solution. On termination, the best solution is put forward as hopefully the optimal SRR. Solution strings are selected by the roulette wheel selection scheme. All prospective parents are given a proportion of the wheel based on their fitness. Using the original fitness function given in Ref. [14], it was found that for large fitness values, the difference in fitness values, tended to be proportionally small. This results in a string with a better fitness having almost the same chance as any other of being selected.

A better fitness function was needed that differentiates well among fitness values, no matter how large the values get. The answer is to normalize the fitness by the "Bestvalue" found in the search so far. In order to highlight the difference in fitness, all values were reduced by the Bestvalue, i.e.

$$x_{Ni} = x_i - \text{Bestvalue} \tag{12}$$

This has its own problem since it allowed negative numbers, not acceptable in probability. The resulting fitness values, F_i, has to conform to the following conditions:

1. $F_i \in [0, \infty]$
2. Negative objective values are to be mapped low.
3. Positive objective values are to be mapped high.
4. The objective value of 0 is to be arbitrarily mapped to a fitness of 1.
5. As objective values increase, the slope of the fitness function must increases, to highlight better answers.

Indeed, the exponential function, $g(x) = e^x$, has all of the properties required. In order to normalize the values obtained, X_{Ni}, is divided by Bestvalue and multiplied by an arbitrary scaling factor, i.e.

$$g(x) = \exp(\text{factor} \times (x - \text{Bestvalue})/\text{Bestvalue}) \tag{13}$$

The exponential function was implemented as a Taylor series:

$$\exp(x) = 1 + x + x^2/2! + x^3/3! + x^4/4! + \cdots + x^n/n! + \cdots \tag{14}$$

For practicality reasons, the polynomial was truncated to the fifth term. However, this truncation results in the polynomial diverging for values of x less than -2. To remedy this, an inverse function, $1/x$, was used for values less than -1. The inverse function was scaled to give the sane slope as e^x at -1. The resulting piecewise function,

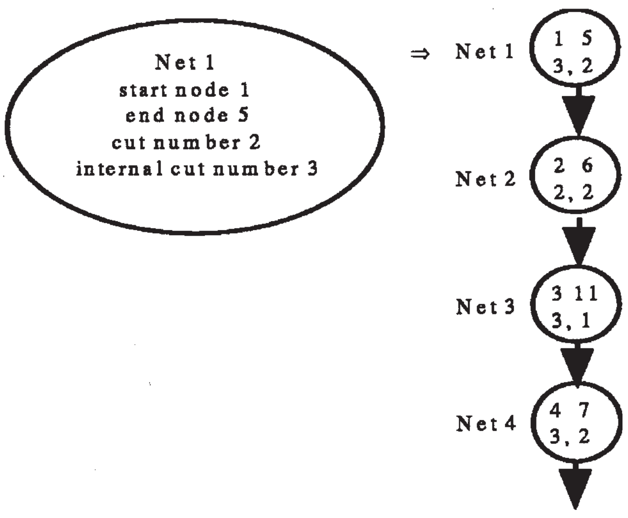


FIGURE 15 Netlist representation.

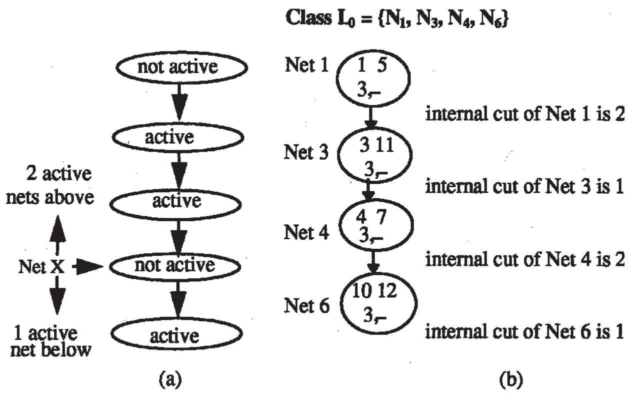

Class L0 = {N1, N3, N4, N6}
not active
active
active
not active
active
2 active nets above
Net X
1 active net below
(a)
Net 1
Net 3
Net 4
Net 6
internal cut of Net 1 is 2
internal cut of Net 3 is 1
internal cut of Net 4 is 2
internal cut of Net 6 is 1
(b)

FIGURE 16 (a) The internal cut at node X after the end node is 2, (b) determining the internal cut numbers of class L_0 in the example in second section of "Some examples of heuristic algorithms".

continuous to the first derivative is:

$$\exp(x) = \begin{bmatrix} 1 + x + x^2/2! + x^3/3! + x^4/4! & x \geq -1 \\ -1/(e \cdot x) & x < -1. \end{bmatrix} \tag{15}$$

In the implementation below, to further reduce computation time, the Taylor expansion was factored, i.e.

$$\exp(x) = \begin{bmatrix} 1 + x(1 + x(1/2 + x(1/6 + x/24))) & x \geq -1 \\ -1/(e \cdot x) & x < -1. \end{bmatrix} \tag{16}$$

Termination

The GA presented previously is terminated under two conditions: either the maximum number of allowed generations has been reached, or the average fitness of the population has not increased by an acceptable amount. The result of using a normalized fitness function is that the fitness values tend to remain static. Rather than increase with time, the average fitness of the population stays relatively constant. This caused the second termination condition to be called too frequently, typically after only 3 or 4 generations, resulting in suboptimal solutions. The answer was to reduce the acceptable increase from 5 to less than 1%.

Tarng *et al.*'s Heuristic

As discussed earlier, the heuristic presented in Ref. [8] is based on cut numbers. To be compatible with the input to the GSRR program Tarng *et al.*'s heuristic needs to initially accept a node list format. This allows the program to use the method of determining cut numbers as discussed in "Results" section. The nodelist then needs to be converted to a net list format, where each net consists of a start and end node and has a cut number and internal cut number as shown in Fig. 15.

After the net list is sorted, based on the nets cut numbers, the internal cut number of each net is found by splitting the net list into classes and then evaluating each class. To evaluate the internal cut numbers of each class, every possible node is searched in turn. Once the start node of a net has been found the net is flagged "active." After the end node of the net has been found, the net is flagged "not active." The internal cut number of each net is then the maximum number of active nets either above or below the net when the start or end node is found. This is shown in Fig. 16.

The function is used for two applications of the heuristic given in Ref. [8]. First, to determine internal cut numbers of each class and finally to evaluate the congestion of the resulting interval graph. A number of issues need to be noted: classes other than the first include previous classes for evaluating internal cut numbers, the residual cut number of a net is redundant as it is only the difference between the cut number and internal cut number for two node nets, and the interval graph representation is built by assigning nets from the middle out. Once the internal cut numbers have been found, the net list is again sorted and the street width at each net found by evaluating the entire net list as a single class.

RESULTS

In this section, we shall examine the speed at which the GSRR is able to generate solutions and the quality of the generated solutions. By applying GSRR to different net lists, we will show it is faster, and can generate better results, than current heuristic methods.

TABLE III Array implementation

Number of nodes (n)	Av. time	Av. time/n	Av. time/($n \log n$)	Av. time/(n^2)
10	204	20.4	20.4	2.1
20	516	25.8	19.8	1.3
40	1221	30.5	19.0	0.8
100	4681	46.8	23.4	0.5
1000	321153	321.1	107.1	0.3

Computation Times for Different Objective Function Implementations

Every time the objective function, "DecodeRouting" is called, a particular routing, (based on a chromosome), is evaluated. This function is the most computationally intense portion of the entire GSRR, and is of critical importance in deciding its complexity. "DecodeRouting" was implemented using both an arrays and lists. While harder to implement, it was expected that the list implementation would have a lower computational complexity, i.e. for a large number of nodes, the list implementation will be quicker.

The following two tables show the average time taken by "DecodeRouting" for between 10 and 1000 node net lists generated at random. It is possible that the solution string may have some part to play in the time taken, so the only chromosomes used consisted entirely of 1's or 0's. All computations were performed on a Sun Sparc 20.

To get an indication of the complexity of the objective function all values were divided by the prospective order. If the resulting value remains relatively constant over the range, then the function is known to be of that complexity.

For example, if n is the number of nodes and the function is really of complexity:

$$T = O(An^2 + Bn + C) = O(n^2) \tag{17}$$

then:

$$\lim_{n\to\infty} \frac{T}{n^2} = \lim_{n\to\infty} \frac{(An^2 + Bn + C)}{n^2} = A \tag{18}$$

where A, B, and C are constants. Thus we know that the example is $O(n^2)$.

As shown in Tables III and IV, three different complexities were tried; $O(n)$, $O(n\ log\ n)$, and $O(n^2)$. $O(n \log n)$ complexity remains linear in the range for the list implementation, so it can be concluded that "DecodeRouting" is $O(n \log n)$. All other computation in the GSRR is of $O(n)$ so the order of the objective function determines the overall complexity.

Figure 17 shows that arrays are faster for a low number of nodes, while lists are faster for a large number. The break even point is about 500 nodes. Since the number of nodes for a given problem remains fixed, CAD package may use both implementations and select the best, based on the number of nodes. All further results will be given in terms of the list implementation.

Optimizing the Fitness Function

As discussed earlier, the fitness function is of the form:

$$g(x) = \exp(\text{factor} \times (x - \text{Bestvalue})/\text{Bestvalue}) \tag{19}$$

The variable *factor* provides the slope of the fitness function. The effect of different slope factors is given in Table V. Obviously a higher slope factor is more severe, implementing an *elitist* selection strategy. In an elitist strategy, only the very best solution strings are breed. On the other hand, a low scaling factor implements a *steady state* strategy, where all strings have a reasonable chance of becoming parents.

The Objective Function and the Quality of Routings Produced

Once the fastest data structure has been implemented, the objective function has a predetermined computational time. However, the effectiveness of the objective function in finding the optimal solution has to be evaluated. The objective of SRR can stated as to maximize:

$$\text{maxcut}_i - Q_0 = \max(c_i) - \max\{C_{\text{us}}, C_{\text{ls}}\} \tag{20}$$

The first objective function tried was the one given in Eq. (20). The results of other objective functions are given in Fig. 18. When the objective function returned the value given in Eq. (20), GSRR (1), it was found that little

TABLE IV Doubly linked list implementation

Number of nodes (n)	Av. time	Av. time/n	Av. time/($n \log n$)	Av. time/(n^2)
10	581	58.1	58.1	5.8
20	1141	57.1	43.9	2.9
40	2286	57.2	35.7	1.4
100	6616	66.2	33.1	0.7
1000	200658	200.7	66.9	0.2

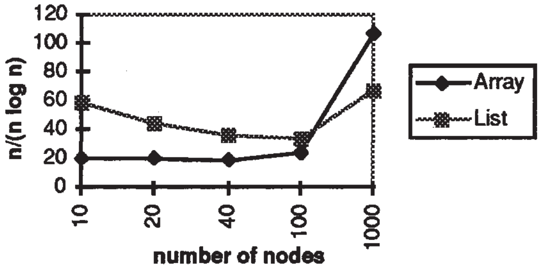


FIGURE 17 Array versus list implementation of DecodeRouting.

TABLE V Different slope factors versus final Bestvalue

Slope factor	Bestvalue obtained
1	470
10	470
50	486
100	490
500	491
1000	494

progress was made in the evolution of a solution. Almost all routings had the same street congestion.

The problem is that large areas of the search landscape have the same street congestion, Q_0. As an example, consider a five-bit optimization problem, where every possible string has the same fitness, except for the string "11111". As the fitness values of all other strings are the same, no bias is given to closer solutions. And so the optimal solution string can only be found at random. For large chromosomes this is unacceptable, as the optimal solution may never be found.

The answer is to smooth the search landscape. Rather than having cliffs between fitness values, we need to smooth out the landscape allowing a more gradual rise to maximums. This can be done by adding more search information to the objective function. By giving extra indications of a chromosome being near a better solution, the selection process can be biased to better chromosomes. In the above five-bit optimization example this extra search information could be the number of 1's in the solution, or the integer the string represents, or the number of times "11" appears, the possibilities are endless.

To avoid these fitness cliffs in the SRR problem, the street width at every node can be considered. Different objective functions were used to evaluate routing, as shown in Fig. 18.

The next objective function used is given by Eq. (21), GSRR (2), in Fig. 18.

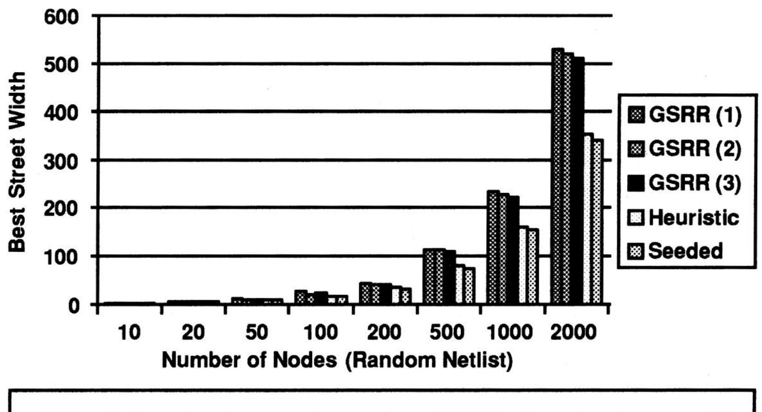
$$\text{maxcut}_i \times \text{size} - \sum q_i = \max(c_i)$$
$$\times n - \sum \max\{c_{i\text{u}}, c_{i\text{l}}\}$$

This objective function consistently produced better routings for all examples tried. Note that a better routing is one with a smaller street width. Other objective functions are possible, but most are similar to Eq. (21).

In GAs many different variables exist that need to be fine tuned. The results given by GSRR (3) in Fig. 18, represent the optimal variables for the objective function in Eq. (21). The optimal conditions found through experiment are:

- Population size, 20 solutions.
- Probability of cross over, $P_{\text{CROSS}} = 0.3$.
- Probability of mutation, $P_{\text{MUT}} = 0.05$.

The heuristic solution in Fig. 18 is that by Tarng *et al.* [8]. As can be seen, the solutions produced are consistently better than the unaided GAs, except in the case of a small number of nodes. When the heuristic is used in the second section of "Some examples of heuristic algorithms", it produces the routing shown in Fig. 6, with street congestion $Q_0 = 3$, as expected. However, the GA



GSRR (1): Fitness based on congestion. **GSRR (2):** Fitness based on street width.
GSRR (3): Different population, mutation rate. **Seeded:** Extended Random Initialisation.

FIGURE 18 Quality of routings produced.

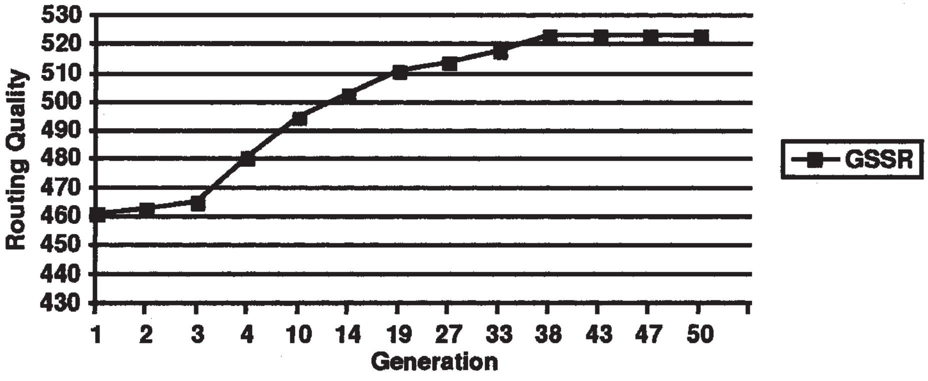


FIGURE 19 A general solution produced by the GSRR.

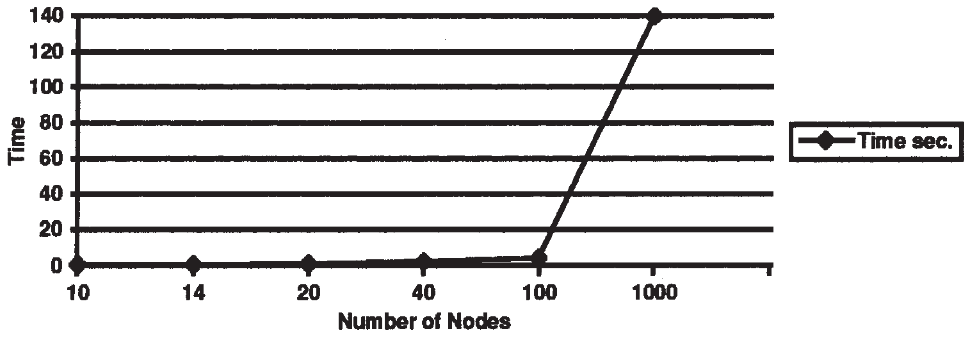


FIGURE 20 Graph of computation time versus the number of nodes in the net list.

almost always produces the routing shown in Fig. 7(a), with street congestion $Q_0 = 2$, a superior routing.

The better solutions produced by the heuristic can be exploited in the GA by implementing extended random initialization. The result of this is shown in Fig. 18, as the "Seeded" solutions. Thus the GA can be used to find improved solutions to the heuristic results.

Computation Times of the GSRR

The GA has been found to converge to a near optimal solution in a relatively short period of time. Figure 19shows a typical graph of a GA converging on a solution. From the observation of many such graphs, the following points have been noted.

- The GA converges to a near optimal solution very fast, after about 50 generations.
- The answers it converges to are normally very good ones, in some cases better than heuristic solutions.
- Once the solutions start to converge, only occasionally does a noticeably better solution appear.

The computation time of the entire algorithm is shown in Fig. 20. The computational time increases as $O(n \log n)$ as expected from the discussion in "Optimizing the fitness function" section.

Another important factor affecting the computation time is the encoding scheme. Varying the length of routing variables increases the search space at the cost of execution time, as shown in Table VI.

As expected, the shorter the chromosome, the shorter the execution time. The time increased at $O(L \log L)$, where L is the chromosome length. The fastest encoding is the one bit per adjacent start node scheme. However, more optimal solutions were not necessarily found in the largest, and slowest, encoding schemes. Two reasons can be given as to why.

1. Large areas of the representation are redundant. If the decision variable calls for more pseudo points than is possible, the extra is ignored. This is another form of fitness cliff, correctable by limiting the value of L.
2. The representation of the decision variables for $L > 1$, enabled *Hamming* cliffs to occur. This can be corrected with *Gray* coding.

Figure 21 shows the number of nodes against the CPU time of the GSRR and Tarng *et al.*'s heuristic. The GSRR

TABLE VI Results for different encoding lengths (time in s)

w	L	Best value	Av. time	Av. time/$(L \log L)$
0	30	488	5	0.11
1	49	498	6	0.07
2	98	488	12	0.06
3	147	490	20	0.06
4	196	454	32	0.07
5	245	404	50	0.09

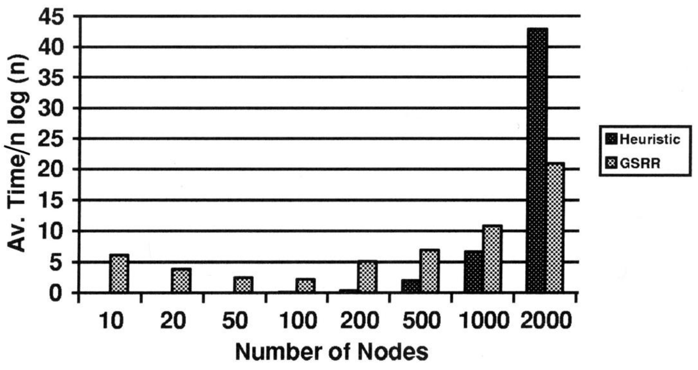


FIGURE 21 Computation times of Tarng *et al.*'s heuristic versus the GSRR.

shown is with a population size of 10 over 50 generations. Thus the GSRR evaluates 500 routing's per execution. It can be noted that the heuristic is very slow for a large number of nodes, as would exist in a practical circuit. The GSRR is faster than the heuristic for more than 500 nodes, (note that the array implementation of GSRR is faster for less than 500 nodes).

In summary, the GSRR developed here is both fast and robust. The array implementation of the objective function was found to be faster for a small number of nodes, while the list implementation is faster for a large number. Changing the *factor* in the fitness function can implement different selection strategies. It was found that the best objective function is one based on street width. The optimal GA has a population size of 20, a probability of cross over of 0.3, a probability of mutation of 0.05 over 50 generations. Tarng's heuristic produces better results than the unseeded GSRR, but the seeded GSRR can produce still better results. Finally, the GSRR is faster than the heuristic for all but a small number of nodes.

CONCLUSIONS

A new approach was developed to solve the SRR problem. The solutions produced are better than the one offered by conventional methods in some cases, as would be expected of an NP-hard problem. The only question that needs to be answered is if the new algorithm is more efficient. After studying the complexity of the code of the GSRR, it was found to be $O(nk)$. Where n is the number of nodes and k is the street width. The street width k, varies with n as follows:

- k is $O(1)$ for a sparse netlist. In this practical case, the GSRR is $O(n)$.
- k is $O(\log n)$ for a random netlist. In this case, the GSRR is $O(n \log n)$.
- k is $O(n)$ for a River routing netlist. In this case, the GSRR is $O(n^2)$.

A River routing netlist is a case where each node needs to be made electrically equivalent with a node symmetric on the node axis. Routing algorithms exist that perform River routing in $O(n)$ time.

This compares with Tarng *et al.*'s heuristic which was found to be of $O(nm)$, where m is the number of nets. For the case of two nodes per net, $n = 2m$, thus the heuristic is always $O(n^2)$. Thus the new algorithm is more efficient except in the River routing case where both are equally efficient.

The new approach is able to produce a lower complexity algorithm because it tackles the problem in a different way. Traditionally heuristics look at the problem from a vertical perspective, resulting in algorithms dependent on m, the number of nets. The new approach looks at the problem from left to right, resulting in an algorithm dependent on k, the street width. This is very similar to Han and Sahni approach [6], but with unlimited street width. Since k is always of lower order than m this approach is more efficient.

Acknowledgements

A.Y. Zomaya thanks the support of the Australian Research Council grants no. 04/15/412/194, 04/15/412/001, and 04/15/031/001. S. Olariu thanks the support of the NSF grant CCR-9522093 and the ONR grant N00014-95-1-0779.

References

[1] So, H.C. (1974) "Some theoretical results on the routing of multilayer printed wiring boards", *Proceedings of the IEEE Symposium on Circuits and Systems.*
[2] Kuh, E.S., Kashiwabara, T. and Fujisawa, T. (1979) "On optimum single row routing", *IEEE Transactions on Circuits and Systems* **26**, 361–368.
[3] Raghavan, R. and Sahni, S. (1983) "Optimal single row router", *Proceedings of the 19th ACM/IEEE Design Automation Conference.*
[4] Raghavan, R. and Sahni, S. (1983) "Single row routing", *IEEE Transactions on Computers* **32**, 209–220.
[5] Tsukiyama, S., Kuh, E.S. and Shirakawa, I. (1980) "An algorithm for single row routing with prescribed street congestions", *IEEE Transactions on Circuits and Systems* **27**, 765–771.
[6] Han, S.Y. and Sahni, S. (1985) "A fast single row routing algorithm", *Proceedings of the 23rd Annual Allerton Conference on Communications, Control, and Computing.*

[7] Han, S.Y. and Sahni, S. (1984) "Single row routing in narrow streets", *IEEE Transactions on Computer-Aided Design* **3**, 235–241.
[8] Tarng, T.T.K., Marek-Sadowska, M. and Kuh, E.S. (1984) "An efficient single-row routing algorithm", *IEEE Transactions on Computer-Aided Design* **3**, 178–183.
[9] Ting, B.S. and Kuh, E.S. (1978) "An approach to the routing of multilayer printed circuit boards", *Proceedings of the IEEE Symposium on Circuits and Systems*.
[10] Liu, L.C. (1987) "Heuristic algorithms for single-row routing", *IEEE Transactions on Computers* **36**, 312–320.
[11] Holland, J.J. (1975) *Adaptation in Natural and Artificial Systems* (University of Michigan Press, Ann Arbor).
[12] Davis, L. (1991) *Handbook of Genetic Algorithms* (Van Nostrand Reinhold, New York).
[13] Goldberg, D.E. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1st Ed. (Addison-Wesley, Reading, MA).
[14] Riberio-Filho, J.L., Treleaven, P.C. and Alippi, C. (1994) "Genetic-algorithm programming environments", *IEEE Computer* **27**, 28–43.
[15] Srinivas, M. and Patnaik, L.M. (1994) "Genetic algorithms: a survey", *IEEE Computer* **27**, 17–26.
[16] Chipperfield, A. and Flemming, P. (1996) "Parallel genetic algorithms", In: Zomaya, A.Y., ed, *Parallel and Distributed Computing Handbook* (McGraw-Hill, New York), pp 1118–1143.

Authors' Biographies

Albert Y. Zomaya is the CISCO Systems Chair Professor of Internetworking. Prior to the current appointment he was a Professor and Deputy-Head in the Department of Electrical and Electronic Engineering at the University of Western Australia. He received his PhD from the Department of Automatic Control and Systems Engineering, Sheffield University, United Kingdom. Also, he held visiting positions at Waterloo University (Canada) and The University of Missouri-Rolla (USA). He is the author/co-author of five books, more than 150 publications in technical journals, collaborative books, and conferences, and the editor of three volumes and three conference proceedings. He is currently an associate editor for the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Systems, Man*, and *Cybernetics* (*Parts A*, *B*, and *C*), *Journal of Parallel Algorithms and Applications*, *Journal of Interconnection Networks*, the *International Journal on Parallel and Distributed Systems and Networks*, *Journal of Future Generation of Computer Systems*, and the *International Journal of Foundations of Computer Science*. He previously served on the editorial board of the *International Journal in Computer Simulation* and the *IFAC Control Engineering Practice Journal*. He is also the Founding Editor of the *Wiley Book Series on Parallel and Distributed Computing*. He is the Editor-in-Chief of the *Parallel and Distributed Computing Handbook*(McGraw-Hill, 1996). Professor Zomaya was elected in July 1999 to Chair the *IEEE Technical Committee on Parallel Processing*. He is also a board member of the *International Federation of Automatic Control* (*IFAC*) *committee on Algorithms and Architectures for Real-Time Control*, and serves on the executive board of the *IEEE Task Force on Cluster Computing*. He served on the steering and program committees of several national and international conferences. Professor Zomaya is the Founding Co-Chair of the Workshop on *Biologically Inspired Solutions to Parallel Processing Problems* (Florida, 1998; San Juan, 1999; Mexico, 2000). He is the Stream Chair of the *10th International Conference on Computing and Information* (Kuwait, 2000), Program Vice-Chair of the *International Conference on Parallel Processing* (Toronto, 2000), General Chair of the *International Symposium on Parallel Architectures, Algorithms*, and *Networks* (Perth, 1999), the Program Vice-Chair of the *2nd International Conference on Parallel and Distributed Computing and Networks* (Brisbane, 1998), Vice-Chair of the *11th International Conference on Parallel and Distributed Computing Systems* (Chicago, 1998), Vice Chair of the *High Performance Computing Conference* (Bangalore, 1997). Professor Zomaya is a chartered engineer and a senior member of the IEEE, member of the ACM, the Institute of Electrical Engineers (UK), the New York Academy of Sciences, the Association for the Advancement of Science, and Sigma Xi. He received the 1997 *Edgeworth David Medal* from the Royal Society of New South Wales for outstanding contributions to Australian Science. His research interests are in the areas of high performance computing, parallel and distributed algorithms, scheduling and load-balancing, computational machine learning, scientific computing, adaptive computing systems, mobile computing, data mining, and cluster-, network-, and meta-computing.

Roger Karpin studied for a Bachelor of Engineering (in Electrical Engineering) and a Bachelor of Computer Science at The University of Western Australia. Karpin's technical interests are in the areas of Internet, Networking, and Parallel Computing.

Stephan Olariu received the M.Sc. and PhD degrees in computer science from McGill University, Montreal in 1983 and 1986, respectively. In 1986, he joined Old Dominion University where he is a Professor of Computer Science. Professor Olariu has published extensively in various journals, book chapters, and conference proceedings. His research interests include image processing and machine vision, parallel architectures, design and analysis of parallel algorithms, computational graph theory, computational geometry, and mobile computing. Professor Olariu serves on the Editorial Board of IEEE Transactions on Parallel and Distributed Systems.