

GM LEARN: AN ITERATIVE LEARNING ALGORITHM FOR CMOS GATE MATRIX LAYOUT

Sao-Jie Chen

Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan 10674, R.O.C.

Yu Hen Hu

Department of Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI 53706, U.S.A.

ABSTRACT

In this paper, an iterative CMOS gate matrix layout algorithm utilizing Artificial Intelligence (AI) learning techniques is presented. This GM_Learn algorithm features a rudimentary learning mechanism which improves the quality of a gate matrix layout through the repetitive application of a one-pass algorithm, called GM_Plan, developed by the authors. Two AI learning paradigms—rote learning and learning by parameter adjustment—are used in GM_Learn to modify the heuristic search parameters based on information of the previous solution. Preliminary results indicate that this novel algorithm is able to produce a high quality gate matrix layout in only a few iterations. Another significance of this method is that it may be applicable to other combinatorial VLSI physical design problems where heuristic guided search is required.

INTRODUCTION

Gate matrix layout is a systematic CMOS layout methodology developed at Bell Laboratories [1]. Given a circuit schematic, a gate matrix layout design maps the MOS transistor gates and input/output nodes into vertical columns using the polysilicon layer. Interconnection, power, and ground wires are realized in horizontal metal wires. Since the locations of MOS transistors are always on the intersections of columns (polysilicon gate) and rows (diffusion channel), this layout style is called a *gate matrix* layout. For convenience, symbolic representations are often used to represent a gate matrix layout (cf. [2]).

The objective of optimization is to minimize the total number of rows (tracks) required in the layout for a given circuit schematic. In designing a gate matrix layout, two basic tasks must be accomplished: the placement of all transistor gates and i/o signal nodes to column slots, and the routing of nets to horizontal tracks. It has been shown that this is an *NP-complete* problem [3], for which, an optimal solution is often impractical. Instead, heuristic approaches are resorted, which lead to suboptimal solutions.

Generally speaking, there are two categories of solutions to the gate matrix layout problem: the *one-pass* approach and the *iterative* approach. The one-pass approach intends to find an acceptable solution in a single trial using the best possible heuristics. Hence, the emphasis is on the efficiency of the algorithm. Examples of the one-pass approach include: "Interval-graph" method [4-7], "net-ordering first" method [8,9], "max-flow min-cut" method [10,11], "dynamic programming formulation" [12], "molecular physics modeling" [13] and "AI planning" [2] methods.

This research is supported in part by National Science Foundation under contract MIP-8896111, and in part by DARPA under contract MDA 903-86-C-0182.

The iterative approach attempts, on the other hand, to improve a given solution by perturbing an existing one so as to minimize some (heuristic) cost function. The emphasis is on the quality of the result. Examples of iterative method include the "pairwise interchange" [14,15] and "simulated annealing" [16]. Both of these iterative methods, being worked on (gate placement and net routing) data directly, are computationally very expensive.

Our objective in this paper is to look for an iterative method which is computationally more efficient than existing iterative methods, and with promising outputs that have better quality than existing one-pass algorithms. To achieve this goal, a novel iterative learning paradigm, called GM_Learn, is developed. A distinctive feature of GM_Learn is the incorporation of an existing one-pass algorithm (GM_Plan), which has a better efficiency, into an iterative learning environment in order to improve the gate matrix layout quality. This is accomplished by developing a learning scheme that utilizes a one-pass algorithm to layout a given circuit schematic several times (iterations). Each time, based on the outcome of the previous run, the learning scheme utilizes the one-pass algorithm slightly differently, by modifying some of its parameters, with the hope that the quality of the present run will be improved. This learning paradigm closely mimics the problem solving strategy of human design engineers who rework on the same problem several times and gradually improve the quality of the design.

REVIEWS OF THE ONE-PASS ALGORITHM: GM_PLAN

GM_Plan is an AI planning-based gate matrix layout algorithm developed by the authors recently [2]. With the use of the hierarchical planning technique, the gate matrix layout problem in GM_Plan is decomposed into subgoals each of which corresponds to the placement of a gate and to the assignment of tracks to nets connecting that gate. Since subgoals interact with each other while competing for track resources, unnecessary tracks may be added due to inappropriate management of subgoal interactions. In GM_Plan, this problem is resolved by using two meta-planning control policies, called Graceful Retreat (GR) and Least Impact (LI). The GR policy gives high priorities to subgoals that have less flexibilities, or are more difficult to accomplish. The LI policy directs the program to look for the solution to a subgoal, among a set of potential solutions, which is the least likely to impact (interfere) the solutions to other subgoals.

In GM_Plan, each subgoal is achieved in four steps: (1) Select a gate to be placed next; (2) Choose a slot to place the gate; (3) Select a set of nets to be routed; (4) Choose tracks to route the selected nets. The second step is the most critical one as it imposes a gate ordering constraint that has profound effect on the remaining solutions. In GM_Plan, this step is accomplished under the control of the LI policy. Specifically, a heuristic cost function is evaluated for each potential slot. As the

cost is lowered, less resources will be required; therefore, there will be less impacts on the remaining solutions. Unfortunately, due to the interaction among subgoals, the cost function can not be evaluated accurately. Some factors of the cost function must be guessed using heuristics. Intuitively, one would conjecture that the more accurate the cost function can be evaluated, the better the gate matrix layout would be. One possibility is to examine results of the previous run(s), and hopefully those previous results will shed some light on the (more) accurate values of the current run. This is exactly the approach taken by GM_Learn as described in the next section.

IMPROVING GATE MATRIX LAYOUT WITH LEARNING

Applications of AI Learning Techniques

One AI learning method used in GM_Learn is an application of the *learning by parameter adjustment method* developed in the Samuel's checkers player program [17,18]. An analogy is drawn between the task of checkers playing and gate matrix layout: Both involve combinatorial search through the solution space where a "good" heuristic move (solution step) relies on the accurate prediction (look ahead) of consequences of the current move. In the checkers player, a linear heuristic cost function is used to evaluate the potential benefit of making a specific move. In the GM_Learn algorithm, a similar linear heuristic cost function (to be described later) is used to determine a slot to place the current gate. Hence, it is natural to use the same learning technique to help improve the accuracy of the cost function estimate.

Another learning method used in Samuel's checker's player [17,18], and hence in GM_Learn, is called "rote learning." Rote learning is a mechanical learning paradigm in which learning means to memorize large amount of raw information without further processing [18]. In the checkers player program, the cost of a move is searched in the stored data base rather than recalculated. Similarly, in GM_Learn, "rote learning" is used to search the information stored in the previous solution.

In order to apply these two AI learning paradigms, a two-step approach is taken: (1) Identify the set of parameters as the object to be learned (what can be learned?); (2) Develop a set of learning rules to adjust the identified parameters in order to produce desired output (how to learn?).

What Can Be Learned?

The heuristic cost function in the one-pass algorithm GM_Plan for slot selection is the object to be learned in GM_Learn. The cost is an estimate of the total wire lengths required when the current gate is placed in a particular slot. According to the Least Impact policy, the slot with the lowest cost is favored as it minimizes the impacts on the routing of nets corresponding to gates which have not yet been placed. This cost function, denoted by, say, f , can further be decomposed into four factors:

$$f = g1 + h1 + h2 + h3$$

Each of these factors deserves some discussion. In GM_Plan, a net will be committed to a track only if most of the gates that are connected to the net have been placed. Once a net is assigned to a track, it is called a *fixed net*. Otherwise, the routing of the net will be postponed until later stages of the design; such a net is called a *floating net*. When a gate is placed into a particular slot, two situations may occur: For those nets that are connected to the current gate, a connection is to be made; the increase in net lengths due to the connection of nets to the newly placed gate is called *connection cost*. The other situation is that the placement of a new gate may cause other existing nets to be stretched; the amount of stretching (expansion) will be called *expansion cost*. Combining the two types of nets with the two situations, we have the four factors contributing to the cost function f : $g1$ is called the *fixed-net connection cost*, $h1$ the *fixed-net expansion*

cost, $h2$ the *floating-net expansion cost*, and $h3$ the *floating-net connection cost*.

From the above discussion, it is clear that the values of the last three factors in the cost function f need to be guessed as they depend on the outcome of future layout steps. If the current solution does not deviate much from the previous one, then the values of these costs functions, evaluated in the previous solution, may serve as better estimates than those obtained using heuristics stated above.

Another type of heuristics which may be improved is the heuristics for initial pruning of unpromising slots to reduce the number of cost function evaluation. In order to improve the search efficiency, GM_Plan does not consider those slots whose neighboring gates do not have direct connection (via one or more nets) with the gate to be placed. Furthermore, using the seed gate as a "continental divide," only one side of the seed gate which has more potential slots will be searched. Yet another type of heuristics which may be improved with an existing solution is the heuristics for assigning nets to tracks. At the routing stage GM_Plan avoids using the track reserved by a fixed net that is not completely connected to its constituent gates. Estimates for both of these heuristics should be benefited by the knowledge of an existing solution.

How to Learn?

Once a set of parameters are identified as the object to be learned, the next question to ask, then, should be:

How to adjust these parameters so as to learn to improve the overall gate matrix layout quality?

Since the relations between the parameters and the output quality are highly non-linear, it is unlikely that an analytical model can be derived to describe the behavior of a gate matrix layout algorithm in terms of those identified parameters. Two extremes can be identified: One may ignore completely the existence of a solution of the same problem, and uses only original heuristics to estimate the values of these parameters as if the solution does not exist. On the other end, one may want to directly substitute values of these parameters from those "learned" from (stored in) the previous solution; this ignores the potential difference between the current solution and the previous solution. Intuitively, some kind of compromise between these two extremes should be more desirable. In GM_Learn, these three types of heuristic rules for adjusting parameters are all implemented and tried. They are called the *forward Euler rule*, the *backward Euler rule*, and the *Trapezoid rule*:

1. *Forward Euler rule*: This is the heuristic which ignores completely the existing solution, and uses only the "guessed" values of the three components, $h1$, $h2$, and $h3$, in the heuristic cost function f .

2. *Backward Euler rule*: This is the other extreme which uses solely the "learned" values of $h1$, $h2$ and $h3$ from the previous solution as the estimates of the current run.

3. *Trapezoid rule*: This heuristic takes a weighted average of the guessed values and the learned (from the previous output) values of $h1$, $h2$ and $h3$.

As expected, after extensive trials and errors, the Trapezoid rule usually gives better performance than other two extreme rules.

The adjustment of the other two types of slot assignment and track routing heuristics can be implemented similarly. In selecting a potential slot, GM_Learn relaxes the heuristics to search only one side of the seed gate so that a better minimum of the cost function may be obtained. In net routing phase, the heuristics which avoids using reserved tracks may be too conservative when the outcome of the previous run is available. Hence, in GM_Learn, this heuristic is also relaxed taking advantage of the availability of the previous solution.

ALGORITHM AND COMPLEXITY ANALYSIS

The main procedure of GM_Learn is outlined in Fig. 1 below. In the code line 2, c is the iteration count; in line 3, oldNO and oldGO are the status tables used to store the information of the previous solution while newNO and newGO are used to store the information generated by the current pass of the process.

```

Algorithm Main:
1 { GM_Plan (NG, GN, oldNO, oldGO);
2   for (i=1 to c)
3     { GM_Learn (NG, GN, oldNO, oldGO, newNO, newGO);
4       oldNO = newNO;
5       oldGO = newGO;
6     }
7   choose the best solution as output.
8 }

Algorithm GM_Plan (NG, GN, NO, GO):
{
  Prepare_Table (NG, GN);           /* Step 0 */
  Unique_pl&route ();              /* Step 1 */
  Form_NNG ();                     /* Step 2 */
  CRGS = NNG;
  Layout ();
  { While (CRGS != NULL) repeat
    { Place_gate ()                  /* Step 3a */
      { select-gate (g, CRGS) with GR policy;
        plan-assign (g, slot, NO, GO) with LI policy;
        update (NO, GO);
      }
      Route_net (N(g))             /* Step 3b */
      if (end of NNG) reroute (N(g));
      PLGS = PLGS ∪ {g};
      CRGS = {CRGS ∪ G(N(g))} \ PLGS;
    }
  }
  Wrap_up_route (all-nets).        /* Step 4 */
}

Algorithm GM_Learn (NG, GN, oNO, oGO, nNO, nGO):
{
  Unique_pl&route ();              /* Step 1 */
  Form_NNG ();                     /* Step 2 */
  CRGS = NNG;
  Layout ();
  { While (CRGS != NULL) repeat
    { Place_gate ()                  /* Step 3a */
      { select-gate (g, CRGS) with GR policy;
        learn-assign (g, slot, oNO, oGO, nNO, nGO)
          with Learning & LI policy;
        update (oNO, oGO, nNO, nGO);
      }
      Route_net (N(g))             /* Step 3b */
      if (end of NNG) reroute (N(g));
      PLGS = PLGS ∪ {g};
      CRGS = {CRGS ∪ G(N(g))} \ PLGS;
    }
  }
  Wrap_up_route (all-nets).        /* Step 4 */
}

```

Fig. 1. GM Learn Algorithm

The above GM_Plan and GM_Learn algorithms differ in plan-assign() and learn-assign() only. Since "plan-assign" uses guess values to calculate $f = g_1 + h_1 + h_2 + h_3$, it needs only the NO and GO tables. The improvement by parameter adjustment algorithm is implemented by "learn-assign" using "learned" values from the oNO and oGO tables and storing the new output information in nNO and nGO.

Since GM_Learn is essentially iterative runs of GM_Plan, its computation complexity can be estimated as the product of the number of iteration runs, denoted by c, and the operations count of GM_Plan. It has been shown [2] that the time complexity of GM_Plan is $O(n^2*m)$ where n is the number of gates, and m is the total number of nets. Thus, the overall computation complexity of GM_Learn should be of $O(c*n^2*m)$.

SIMULATION RESULTS WITH GM LEARN

Two sets of benchmark circuits were run to test GM_Learn and promising results were generated as shown in Tables 1 and 2. Legends used in these two Tables are: The "lower bound" (column 1) equals to the maximum number of nets per gate in each circuit; If the number of tracks used in the layout is equal to the lower bound, an optimal solution is obtained (marked by an "*"); Known solutions (col. 2) are the best solutions found in the literature; Column 3 gives the planning solutions after the first iteration; Column 4 gives the learning solutions after the six iteration; The final solutions of GM_learn (col. 5) will be chosen from the best of the planning and learning solutions; Column 6 contains the CPU time of GM_Learn after the sixth iteration on a VAX 11/750 computer.

The first set of circuits in Table 1 are collected from published literature (cf. [2]). Most of the solutions are optimal at the first time by GM_Plan (7 optimal solutions out of 16 examples) or turn out to be optimal by the learning adjustment (10 out of 16 examples). Only two learning examples converge with solutions of a higher track number (enclosed by parentheses in Table 1), for their first planning solution is good enough and cannot be improved anymore. In total, GM_Learn delivers the best (11/16 optimal) solutions by combining both the planning and learning solutions.

The run times for "w3" and "w4" cannot be obtained for the reason of "memory limit exceed" on a "VAX 11/750" with only 4 Mbytes of memory. The results shown are obtained from running the program on a "Sequent Symmetry S81" computer, which is faster and has larger memory (32 Mbytes).

Table 1.— Layout Results with GM Learn.

Name	1	2	3	4	5	6
v4000	5	7	6	6	16.7	
v4050	5	6	5*	5*	13.1	
v4090	9	11	10*	10*	43.1	
v4470	5	14	11	11	73.2	
vc1	9	9*	10	9*	9*	36.1
v1	3	3*	3*	3*	3*	3.2
vw1	4	4*	4*	4*	4*	3.1
vw2	3	5*	5*	5*	5*	4.9
w1	4	4*	4*	4*	4*	12.8
w2	14	14*	14*	14*	14*	66.8
w3	11	23	21	(23-25)	21	N/A
w4	18	36	46	34	34	N/A
wan	6	6*	6*	6*	6*	4.1
wli	4	4*	6	4*	4*	6.4
wsn	4	8*	8*	(10)	8*	23.6
x0	6	15	15	13	13	102.0

(* means optimal solution; column 1 is the Lower Bound; 2 is the Known Solution; 3 is the planning Solution; 4 is the learning Solution; 5 is the "planning + learning" Solution; 6 is the CPU Time on VAX 11/750. The measure units used for col. 1-5 are "tracks", that of 6 is "second".)

Table 2.—Special Gate Matrix Layout Results.

Name	1	2	3	4	5	6
x1	2	5*	5*	5*	5*	6.7
x2	2	6*	6*	6*	6*	13.2
x3	2	7*	7*	7*	7*	24.0
x4	2	2*	2*	2*	2*	3.1
x5	2	2*	2*	2*	2*	8.3
x6	2	2*	2*	2*	2*	24.4
x7	3	4*	6	4*	4*	4.9
x8	3	4*	10	4*	4*	15.5
x9	3	4*	18	4*	4*	58.6

The second set of examples includes some special cases from [12] which were used to evaluate whether an algorithm is greedy. This set of outputs (cf. Table 2) gives 6/9 optimal solutions for the planning method, and 9/9 for the learning one. As shown in Table 2, while GM_Plan generates the worst case solutions for the last three examples (x7, x8 and x9), GM_Learn delivers optimal solutions. The total improvement made by the learning method in these two set of examples is approximately 28% (20/25 - 13/25).

GM_LEARN PERFORMANCE EVALUATION

Since the gate matrix layout is NP-complete by itself and many of the solutions generated by GM_Learn are "absolute approximate" (which seems impossible for ordinary greedy or on-line algorithms,) or better than known solutions, it seems that GM_Learn can be named an "absolute approximate" algorithm. But this cannot be true as Deo has claimed in [12] that "unless $N = NP$, no polynomial-time absolute approximation solution can be found." Nevertheless the GM_Learn method is undoubtedly "relative approximate," for it is far better than the general "greedy" and "on-line" algorithms presented in [12].

Moreover, the GM_Learn method, though with a complexity of $O(c^*n^2*m)$, converges very fast; most of the test runs converge at the second learning iteration and some within the sixth run. Almost all of the convergent outputs are optimal or at least better than known solutions, for at each iteration the same planning techniques are used to avoid as many premature commitments (which cause local optimization) as possible.

This compares favorably with other iteration methods, such as the "pairwise interchange" method [14], which will take $O(n^2(n-1))$ time to achieve an optimal solution, or the "Simulated Annealing" method [16], which will also spend a long computation time before its cooling down.

CONCLUSION

In this paper, GM_Learn, an improvement by learning algorithm, is presented. The promising results obtained from GM_Learn result from the many improvement techniques included in it:

1. The efficient one-pass GM_Plan used as the base algorithm gives a good start.
2. The basic *rule learning* paradigm helps to generate the "learned values" (rather than the "guessed values") that increase the accuracy of the cost function.
3. The *forward Euler*, *backward Euler* and *trapezoid* ideas from digital control research serve as the adjustment mechanisms in our improvement by learning (or with feedback) algorithm. Among the simulation results, the *trapezoid* adjustment is observed to be the best mechanism for most of the inputs.
4. The *heuristic relaxation* method is used to avoid the drawback of the inherent "greedy" feature of a heuristic cost function based algorithm. This method is powerful, for it gives GM_Learn a potential to skip the local optimal point and continue to climb up toward a global optimal.

This will help to conclude that the "improvement by learning" method provides an advancement and can serve as the basis of applying more sophisticated learning techniques for improving VLSI layout.

REFERENCES:

- [1] A. D. Lopez and H-F. S. Law, "A Dense Gate Matrix Layout Method for MOS VLSI," *IEEE Tr. Electron Devices*, vol. ED-27, no. 8, pp. 1671-1675, Aug. 1980.
- [2] Y. H. Hu and S. J. Chen, "GM Plan: A Gate Matrix Layout Algorithm Based on AI Planning Techniques," *Proc. ISCAS*, 1989.
- [3] T. Kashiwabara and T. Fujisawa, "An NP-Complete Problem on Interval Graph," *Proc. ISCAS*, pp. 82-83, 1979.
- [4] T. Ohtsuki, H. Mori, E. S. Kuh, and T. Fujisawa, "One-Dimensional Logic Gate Assignment and Interval Graph," *IEEE Tr. Circuits and Systems*, vol. CAS-26, no. 9, pp. 675-684, Sep. 1979.
- [5] O. Wing, "Automated Gate Matrix Layout," *Proc. ISCAS*, pp. 681-685, 1982.
- [6] J. T. Li, "Algorithms for Gate Matrix Layout," *Proc. ISCAS*, pp. 1013-1016, 1983.
- [7] O. Wing, S. Huang and R. Wang, "Gate Matrix Layout," *IEEE Tr. CAD*, vol. CAD-4, no. 3, pp. 220-231, July 1985.
- [8] T. Asano and K. Tanaka, "A Gate Placement Algorithm for One-Dimensional Arrays," *J. of Information Processing*, vol. 1, no. 1, pp. 47-52, 1978.
- [9] S. Huang and O. Wing, "Improved Gate Matrix Layout," *Proc. ICCAD*, pp. 320-323, 1986.
- [10] C. K. Cheng, "Decomposition Algorithms for Linear Placement and Application to VLSI Design," *Proc. ISCAS*, pp. 1047-1050, 1985.
- [11] D. K. Hwang, W. K. Fuchs, and S. M. Kang, "An Efficient Approach to Gate Matrix Layout," *IEEE Tr. CAD*, vol. CAD-6, no. 5, pp. 802-809, Sep. 1987.
- [12] N. Deo, M. S. Krishnamoorthy, and M. A. Langston, "Exact and Approximate Solutions for the Gate Matrix Layout Problem," *IEEE Tr. CAD*, vol. CAD-6, no. 1, pp. 79-84, Jan. 1987.
- [13] Y. C. Chang, S. C. Chang, and L. H. Hsu, "Automated Layout Generation Using Gate Matrix Approach," *Proc. 24th DA Conf.*, pp. 552-558, 1987.
- [14] H. Yoshizawa, H. Kawanishi, and K. Kani, "A Heuristic Procedure for Ordering MOS Arrays," *Proc. 12th DA Conf.*, pp. 384-393, 1975.
- [15] M. L. Yu and W. J. Kubitz, "A VLSI Cell Synthesizer with Structural Constraint Considerations," *Proc. ICCAD*, pp. 58-60, 1985.
- [16] H. W. Leong, "A New Algorithm for Gate Matrix Layout," *Proc. ICCAD*, pp. 316-319, 1986.
- [17] A. L. Samuel, "Some Studies in Machine Learning with the Game of Checkers," *IBM J. Res. Develop.*, 3, pp. 210-220, 1959.
- [18] A. Barr and E. A. Feigenbaum eds., *The Handbook of Artificial Intelligence, Volume III*, William Kaufman Inc., New York, ch. XIV, 1982.