PHYSICAL DESIGN: MATHEMATICAL MODELS AND METHODS
KEYNOTE ADDRESS

T. C. Hu
Department of Computer Science and Engineering
School of Engineering
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093
hu@cs.ucsd.edu

The title of this talk is too broad to be covered in one day and I realize that there are many experts of different backgrounds, as well as students, in the audience. Thus, I have selected a few basic concepts and two numerical examples to supplement my slides.

1. PROBLEM OVERVIEW

First, the goal of physical design is to obtain a good design, a design that achieves maximum performance and reliability, with minimum area and cost. Since some of the criteria are in conflict, we need a balanced design.

In mathematical programming terms, we have a multi-objective or several objective functions. One way to handle multiple objectives is to use a weighted objective function with the weighting factor proportional to the importance of the term (say, speed or area). During the optimization process, if the speed of the chip is fast enough, we can delete speed as an objective and maintain speed as a constraint.

Second, we usually do not have the time to design every cell in a tailor-made fashion; it is much cheaper and less error-prone if we build the design from a cell library. The situation is similar to building a house where windows have standard sizes. In other words, we restrict our variables to have only limited, fixed values. In the extreme case, we want the variables x_j to be zero or one. Or, we may want all solutions to be positive integers and not arbitrary real values. In design terms, we need "modular" designs. In short, we need integer programming.

Third, the ideal algorithm will first give us a design that satisfies all specifications, and then allow us to gradually improve the objective function if time permits. In mathematical programming terms, we need a primal method.

2. SOME MATHEMATICAL PROGRAMMING TOOLS

Having said all this, I would like to introduce intuitively the following three topics in mathematical programming:

1. Shadow prices in linear programming (LP).
2. The column generating technique in linear programming.
3. The handshake algorithm in integer programming (IP).

ISPD'97 Napa Valley, California USA

(How to stop worrying about P=NP? and start solving integer programs.)

2.1. Shadow Prices

In routing, we need to connect many nets, connecting pins in different regions of the chip. If a given channel is very crowded, we try to avoid that channel and go around by using several relatively empty channels, just like we avoid driving through a busy city block. Intuitively, we could assign a cost to traversing a city block that is proportional to the amount of traffic in that block, in which case we want to find the route to the destination of minimum total cost.

The cost associated with the block or the channel is the *shadow price* of that channel. The shadow price is zero when the channel is empty, and keeps changing as we gradually assign nets in the channel. However, the correct price to be assigned to a given channel does not depend only on the number of nets in that channel — it also depends on the "traffic" in the neighboring channels, and indeed the neighboring channels of neighboring channels.

In LP we have a square matrix B reflecting the current routing, while the shadow price is obtained by multiplying the current cost by the inverse matrix B^{-1}. The inverse matrix B^{-1} not only gives the shadow price but also records what we have done.

2.2. Column Generating Techniques

A typical criticism against using linear programming in solving CAD problems is that the LP matrix is too large. If every column of the matrix denotes a possible activity then there may be thousands or millions of columns. When it is impossible even to write down all the possibilities, how can we possibly solve the problem?

This brings us to the second topic: column generating techniques. Is there a way to obtain the optimum solution to a LP when we cannot write down all the columns of the matrix?

The answer is "no," if all columns can have arbitrary coefficients. In that case we must test every column to check if it can provide some improvement. However, in a real world problem, every column of the matrix means something. For example, a column may represent a possible way of routing.

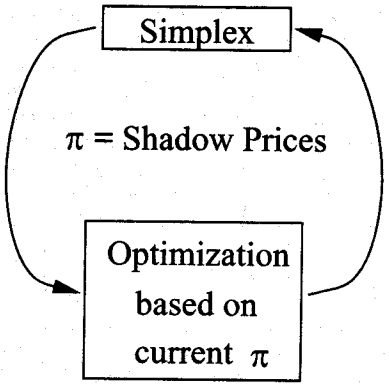
There is an implicit "feasibility rule" to check if a column belongs to the LP matrix. For example, suppose a column consists of six elements and every element represents a channel on the chip. We are not interested in an arbitrary set of six channels: we only want six channels which can form a path connecting the driver output to the receiver input. Thus, even if we cannot write down all the columns, we have in fact implicitly defined all possible columns. Using

the feasibility rule, we can generate a column to improve the current solution.

When the current solution cannot be improved by using the "best" column among all possible columns, it is an optimum solution. Here, the best column is the minimum cost column in terms of the current shadow prices.

Conceptually, we partition a large LP into two parts. One part is a regular LP with a square matrix that generates the shadow prices. The other part then uses the current shadow prices to discover the best column and supply it back to the first part. This is shown symbolically in Figure 1. The top part of the cycle in the figure generates the shadow prices which reflect the current solution. The bottom part is another optimization problem: finding the best column in terms of the current shadow prices. Once we find this column, it is shipped to the top part to improve the current solution. If the best column yields no improvement, then no column can, and the current solution is optimal.

Figure 1.



2.3. Group Minimization in Integer Programming

Let us illustrate the concept of integer programming by a small example where the first two columns are the basic columns, and all other columns are nonbasic columns.

$$\min Z = 2x_1 + 2x_2 + 8x_3 + 8x_4 + 8x_5 \qquad (1)$$

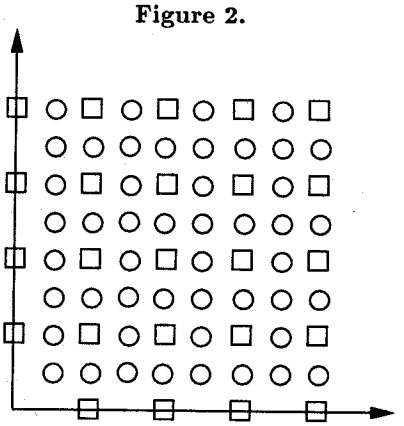
subject to

$$x_1\begin{bmatrix}2\\0\end{bmatrix} + x_2\begin{bmatrix}-0\\2\end{bmatrix} + x_3\begin{bmatrix}3\\3\end{bmatrix} + x_4\begin{bmatrix}4\\3\end{bmatrix} + x_5\begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}b_1\\b_2\end{bmatrix}$$

When both components of the RHS are even integers, say $b_1 = b_2 = 20$, then the optimum solution is guaranteed to have integer components. In other words, the RHS can be expressed as an integer combination of the basic columns. When one or both components of the RHS are odd integers then we need to increase the value of some nonbasic variable from zero to a positive integer. This is shown in Figure 2 where the squares denote $[b_1, b_2]$ values for which the LP solution is guaranteed to be an integer solution.

The general approach is to multiply every column (basic or nonbasic) and the RHS by the inverse matrix B^{-1} and

Figure 2.



then treat the constraint equations as a congruence relation mod 1 (ϕ). In other words, we use the operator ϕB^{-1} to map every column into a group element, with all basic columns mapped into the zero element of the group.

Next, consider a second example IP.

$$\min Z = 4x_1 + 3x_2 + 9x_3 + 9x_4 + 9x_5 + 9x_6 \qquad (2)$$

subject to

$$x_1\begin{bmatrix}3\\1\end{bmatrix} + x_2\begin{bmatrix}1\\2\end{bmatrix} + x_3\begin{bmatrix}4\\2\end{bmatrix} + x_4\begin{bmatrix}5\\3\end{bmatrix} + x_5\begin{bmatrix}2\\1\end{bmatrix} + x_6\begin{bmatrix}3\\2\end{bmatrix} = \begin{bmatrix}12\\12\end{bmatrix}$$

$$B = \begin{bmatrix}3 & 1\\1 & 2\end{bmatrix}$$

$$B^{-1} = 1/5\begin{bmatrix}2 & -1\\-1 & 3\end{bmatrix}$$

In this example the first two columns are the basic columns in the LP, and all nonbasic columns are mapped into group elements g_1, g_2, g_3, g_4. For example, the third column is a non-basic column and is mapped into the group element g_1.

$$\phi B^{-1}\begin{bmatrix}4\\2\end{bmatrix} = 1/5\begin{bmatrix}1\\2\end{bmatrix}$$

$$\phi B^{-1}\begin{bmatrix}5\\3\end{bmatrix} = 1/5\begin{bmatrix}2\\4\end{bmatrix}$$

$$\phi B^{-1}\begin{bmatrix}2\\1\end{bmatrix} = 1/5\begin{bmatrix}3\\1\end{bmatrix}$$

$$\phi B^{-1}\begin{bmatrix}3\\2\end{bmatrix} = 1/5\begin{bmatrix}4\\3\end{bmatrix}$$

There are at most five group elements since $\det B = 6$ (the group is of order $|\det B| - 1$), and the group relationship becomes:

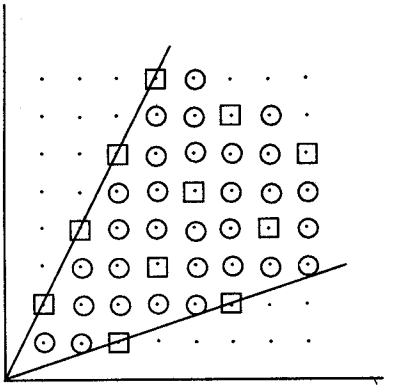
$$x_3 g_1 + x_4 g_2 + x_5 g_3 + x_6 g_4 \equiv g_2.$$

With this approach, even if the LP matrix has millions or billions of columns (more than could possibly be listed), we only need to solve a congruence relation (mod 1) where the number of group elements is at most $|\det B| - 1$.

In Figure 3, the square points again indicate the RHS values which can be expressed as positive integer combinations of the basic columns. Different circular points represent different group elements. Since the right hand side can

only be one of those circular points, we need only solve a group congruence relation in order to discover the correct combination of nonbasic columns to reach the circular point (representing the RHS) from a given square point.

Figure 3.



The algorithm consists of the following steps (see [2]):

1. Solve the IP as if it were a LP.
2. Multiply every nonbasic column and the RHS by B^{-1} and delete the integer part of every column; *i.e.*, map every column by the operator (ϕB^{-1}).
3. Map all nonbasic columns and the RHS into group elements, then solve the congruence relation.
4. Substitute the results back into the original equations and check if all basic variables are nonnegative.

To solve the group minimization problem where every group element has a non-negative cost, we can use the handshake algorithm (see [2] for details).

There are two kinds of labels: temporary and permanent. Initially, all group element labels are temporary. The algorithm repeats the following steps:

1. Among all group elements with temporary labels, change the minimum-cost temporary label to permanent.
2. Perform pairwise addition of the cost of the element whose label was just made permanent with every other element with a permanent label (like a new member of a social club shaking hands with every old member).

Each sum gives a cost for the group element derived by combining the pair. We update the cost of that element if the derived cost is less than the existing cost.

2.4. Applicability of Group Minimization

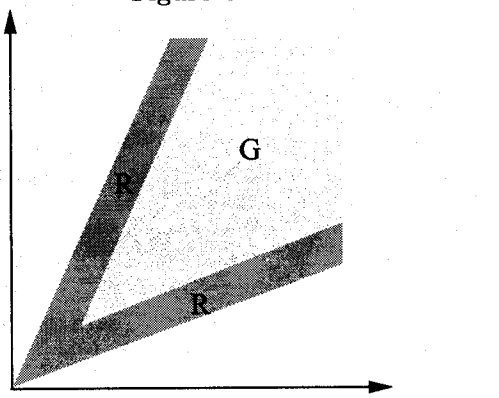
We summarize the application of group minimization as follows. Recall that the basic vectors of a linear program generate a convex cone. If the right-hand side (RHS) does not lie inside the cone, then there is no feasible solution.

Among the points with integer coordinates inside the cone, some points can be expressed as integer combinations of the basic vectors. If the RHS is one of those points, then we have an integer solution automatically. If the RHS is not one of those points, then we need to use nonbasic vectors. To find the cheapest combination of the nonbasic vectors, we solve a congruence relation which requires $O(D^2)$ time where D is the value of the determinant of the basic matrix. We then substitute the values of the nonbasic variables obtained from the congruence relation into the original constraint equations of the LP and solve for the basic variables.

If the basic variables are positive, then we have the optimum integer solution. If any of the basic variables are negative, then we need to use backtracking or some other method which could take $O(2^n)$ time. However, there is an upper bound on the sum of nonbasic variables satisfying the congruence relationship. Geometrically, this bound means that the addition of nonbasic vectors can shift any point from its original position by a maximum distance of δ or less.

In Figure 4, we show a convex cone with the interior of the cone colored green and the boundary layer of the cone colored red. The thickness of the boundary layer is δ.

Figure 4.



Any RHS within the green area can be solved by the congruence relation without causing the basic variables to become negative. If the RHS is in the red area, then the basic variables may become negative.

If we draw a circle centered at the origin with radius r, then some points on the circle are green and some are red. The larger the radius r, the higher the proportion of green points to red, since the thickness δ of the red stripe is a constant.

This means that most integer programs (each corresponds to a possible RHS) can easily be solved by the congruence relation. This beautiful theorem was discovered by Dr. R. E. Gomory more than 30 years ago. At the time of its discovery, the congruence relation was solved by a knapsack algorithm that takes pseudo-polynomial time. I wrote the paper [2] that points out that the congruence relation can be solved in $O(D^2)$ time (the handshake algorithm).

Dr. R. Tarjan used amortized analysis to show that the average cost of an operation in an algorithm can be small, even if the worst-case cost is high. Here we have shown that the average time to solve an arbitrary instance of an integer program (any possible RHS) is polynomial. As you know, integer programming is a known NP-complete problem. That may explain why we are quite successful in physical design even though many subproblems in physical design are NP-complete or NP-hard.

3. CONCLUSION

In conclusion, the speed of computers is getting faster and faster, and the cost of computing, cheaper and cheaper. The trend of "faster and cheaper" means that we can afford to do complicated computations using sophisticated mathematical techniques.

Of course, algorithms are only one cornerstone of design. The real challenge is to understand the real world design problem, to construct a relevant mathematical model, to invent an appropriate algorithm, and to implement it with clear data structures and programming.

This requires cooperative effort involving exchange of ideas and knowledge among a team of members with different backgrounds and expertise, and that is why we are all here.

Thank you very much for your attention.

REFERENCES

[1] HU, T. C. *Integer Programming and Network Flows.* Addison-Wesley, Reading, MA, 1969.

[2] HU, T. C. On the asymptotic integer algorithm. *Journal of Linear Algebra and its Applications 3*, 3 (1970).

[3] HU, T. C., AND KUH, E. S., Eds. *VLSI Circuit Layout: Theory and Design.* IEEE Press selected reprint series. IEEE Press, New York, 1985.

[4] KAHNG, A. B., AND ROBINS, G. *On Optimal Interconnections for VLSI.* The Kluwer international series in engineering and computer science. Kluwer Academic Publishers, Boston, 1995.

[5] LENGAUER, T. *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley & Sons, New York, 1990.

[6] MICHELI, G. D. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, New York, 1994.