22 Jun 1970

A Simplification Heuristic For Large Flow Tables

R. J. Smith

James H. Tracey
Missouri University of Science and Technology

A SIMPLIFICATION HEURISTIC FOR LARGE FLOW TABLES

by:

R.J. SMITH II
Missouri Institute of Psychiatry
University of Missouri School of Medicine
5400 Arsenal Street
St. Louis, Missouri 63139

and

J.H. TRACEY
Electrical Engineering Department
University of Missouri--Rolla
Rolla, Missouri 65401

SUMMARY

Flow tables specifying large asynchronous sequential circuits often contain more internal states than are required to specify desired circuit behavior. Known minimization techniques appear unsuited for reduction of such large (rows x columns $\geq$ 250) flow tables, because of excessive computation and intermediate data requirements for problems of this size. The algorithm described here is intended to rapidly produce a simplified--but in general non-minimal--flow table. It is most economical when applied to extremely large tables and was devised primarily for automated design applications.

The procedure has been programmed in PL/1 and has been incorporated into an asynchronous sequential circuit design automation system[1] developed at the University of Missouri--Rolla. Typical flow table simplification times obtained using the program are cited. In one test reduction of a 217 row x 8 column table to 39x8 required about 2.6 minutes (the minimum table in this case was known to be 23x8).

Introduction

The operation of a sequential circuit is often described by means of a flow table. The flow table columns represent input states, while rows represent internal states assumed by the machine. Each flow table entry specifies the next state resulting from a given input and internal state.

Flow tables, produced either by circuit designers or design automation routines, often contain more internal states than are required to specify the desired circuit behavior. It is advantageous to reduce the flow table to more compact form in such cases, for synthesis costs and circuit complexity increase with flow table size.

Studies of the flow table minimization problem[2,3,4] have produced several algorithms which appear to be applicable only to rather small (less than 15 internal states) flow tables. All seem unsuitable for the reduction of large (100 or more row) tables, for they require an amount of effort and intermediate data estimated to be roughly proportional to the sixth power of the number of rows.

The algorithm presented in this paper is intended to rapidly produce a simplified--but in general non-minimal--flow table. The table to be simplified is assumed to be incompletely specified, and the procedure is most effective if many next state entries are unspecified. The technique is designed to be most economical when applied to extremely large (up to several hundred row) flow tables, and is intended primarily for automated design applications (see, for example, the system described in reference 1).

Two important considerations guided the development of the reduction procedure. First, exhaustive computations or comparisons should be avoided in order to obtain economical reduction times for extremely large flow tables. Second, due to digital computer main memory size restrictions, a modest amount of data (compared to minimization techniques) should be utilized by the flow table simplification heuristic.

These two constraints have led to the adoption of a simple strategy: Only a single set of compatibility classes (CC's), representing the reduced table, is generated. Cover is insured by insisting that each state of the original machine be a member of one and only one compatibility class. Closure is preserved by continuously updating next state and output specifications for flow table rows representing each compatibility class. ("Compatibility class" as used here refers to a class containing two or more rows from the original table.) Closure requirements at any point in the reduction procedure thus reduce to satisfying compatibility requirements for states of the partially reduced machine.

Next state entries for the table to be simplified are stored in a two-dimensional array, with each row used to store the entries for a single stable state; columns correspond to input states. A Boolean matrix is utilized to store output states associated with stable states. As simplification proceeds, compatibility classes (CC's) are formed from sets of states from the original table. Each CC is represented as a flow table row, and is stored in an array position previously occupied by one of the compatibility class members.

A tag is associated with each flow table row: if zero, the tag indicates that the row appeared in the input table and is not a member of any CC. If the tag is negative, the corresponding flow table row is a compatibility class containing two or more rows of the original table. A positive tag indicates that the corresponding row is in a CC which is stored elsewhere; the arithmetic value of the tag indicates the row number in which the CC is located. Note that any row with a positive tag may be ignored in all further processing, since positive tags map original machine states into compatibility classes, and eventually into states of the reduced machine.

Figure 1 shows a portion of a partially reduced flow table and the corresponding data representation.

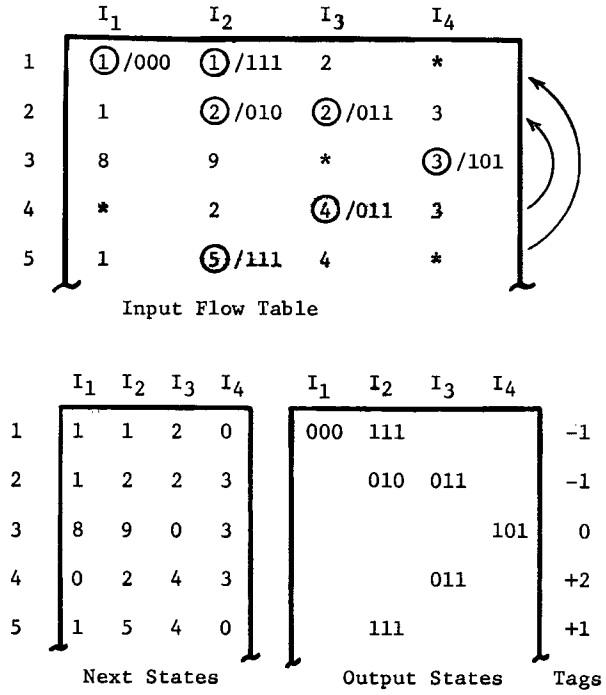


Figure 1. Flow Table and Corresponding Representation.

Next state zero indicates that the flow table entry is not specified. Row 4 has been combined with row 2, and the resulting compatibility class has been stored in row 2; likewise, rows 1 and 5 have been combined and the resulting class is in row 1.

Compatibility Class Computation

During simplification, each state of the original flow table is considered in turn. To reduce the number of row-pair comparisons performed, row i is compared only with compatibility classes--or rows--in the limited range $(i-p) \leq i \leq (i+q)$. The limited flow table examination range may be visualized as a "window" which moves down the flow table. Only rows currently "exposed" in the window are used in flow table simplification.

Because of the "look-ahead" beyond row i, the current status of row (i+q) is required each time the window moves. Thus prior to examining row i from the original table, tag numbers are used to map next state entries for original row (i+q) into compatibility class references if appropriate.

To decrease the amount of intermediate data involved in flow table simplification, only four simple types of row-pair compatibility conditions are utilized; this approach also improves operating speed of the simplification procedure.

Compatibility Class Expansion.

An attempt is first made to add row i of the original table to a CC (tag negative) within the examination range. Since the compatibility class j is represented by its resulting flow table row j, i can be added to class j if the two rows are compatible ($i \sim j$). Two flow tables rows are compatible if for each input state having specified next states in both rows, 1) both next state entries are identical, or 2) both next state entries are stable states whose output states agree whenever both are specified, or 3) the next states are themselves compatible. The third condition has been discarded in this work in order to develop an economical simplification heuristic.

Row i is immediately added to the first compatibility class j with which it is compatible. The resulting compatibility class has a stable state and output specification wherever either of the previous rows was stable. If both were stable for some input, only the outputs are combined. For convenience, the new class is placed in the same location as the old compatibility class; this practice generally reduces the number of next state entries which must be changed, since next state i is likely to appear less frequently than j. The tag for row i is set equal to j, and known (i.e. within the range "window")

next state entries corresponding to stable states i are changed to j.

Next, an attempt is made to add each lower (k>j) compatibility class to the new class containing state i. Any classes which can be added to the new class j are included immediately by updating the appropriate tag, next state and output entries.

Finally, the new class j is checked for compatibility with any single member classes--rows which have previously been found incompatible with all others in the know segment of the reduced table.

The new compatibility class expansion procedure causes compatibility classes which may contain many original table rows to grow quite rapidly; this is advantageous because it quickly decreases the size of the partially reduced table and thus reduces the number of row-pair comparisons performed in each step.

New Class Formation.

If a newly considered flow table row is incompatible with all known compatibility classes (i.e., those in range) with two or more elements, an attempt is made to combine that row with each known single element compatibility class. These classes correspond to rows of the original flow table which have been found incompatible with all known rows. If a single row j is discovered to be compatible with i, a new compatibility class is formed and recorded in the old compatibility class position j as outlined above; the remaining single element classes are also checked for compatibility with new class j, as in the previous new class case.

It has been found that for partially reduced incompletely specified flow tables, pair (i,j) is sometimes the only implicant for class (m,n). In this case, the compatibility of pair (i,j) implies that of pair (m,n). However, this situation does not occur frequently enough to justify rechecking the compatibility of each row pair after formation of each new compatibility class. To do so would increase manyfold the amount of effort expended in flow table reduction.

It can be shown, however, that in general only a small fraction of row pairs need to be rechecked. Furthermore, these pairs can be easily located during the process of next state entry updating after formation of the new compatibility class (i,j).

If row-pair (i,j) is an implicant of pair (m,n), then both i and j must have stable states under some input state(s), and for at least one of these inputs, both i and j must appear as explicit next state entries in rows m and n. If (m,n) implies (i,j), then i and j must be next state entries under at least one input state of pair (m,n). Normal mode operation requires that transitions lead directly to stable states, so both i and j must be stable for the given input.

It is clear that the above theorem dramatically reduces the amount of rechecking which needs to be done after compatibility class formation. Rechecking does, however, represent a significant increase in computational effort, and should be further justified.

First consider two relatively small compatibility classes i and j. In a large table, it is quite likely that the number of unstable next state entries leading to them will be small. Rechecking in this case is inexpensive, especially since the number of stable states per row may be small, further reducing the likelihood of both being stable in the same column.

If on the other hand i and j are large compatibility classes having many stable state columns, rechecking may involve a large number of row-pairs and thus become less desirable.

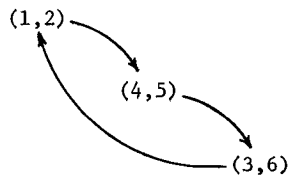
In the formation of new compatibility classes described above, at least one of the constituents of the new class is always a single row from the original flow table, i. Rechecking is performed after formation of new classes resulting from the construction of class (i,j). If rechecking discovers compatible state pairs (m,n) they are immediately combined, but further rechecking based on these "secondary" new classes is not performed. Experimental simplification of large randomly redundant tables has shown that locating implicants of secondary new classes, although costly, results in little if any increase in overall flow table simplification.

Chaining.

A row i from the original table is not considered further unless, after the processes described, it is found to be incompatible with all known (in range) classes. Since the amount of reduction achieved may be significantly decreased by such rows, another attempt is made to find compatibility classes containing i.

A single implication chain consists of a collection of state pairs such that each pair of states (excluding perhaps the last) is conditionally compatible and implies only the next pair in the chain.

Figure 2 shows a partial flow table containing a single implication chain.



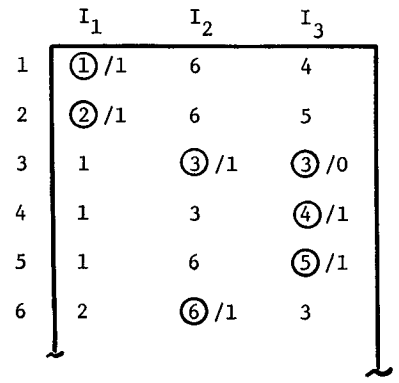


Figure 2. A Single Implication Chain and Partial Flow Table.

Since the generation and use of implication data is expensive in terms of both storage and computation, only single implication chains are used in the flow table simplification heuristic. Additional constraints restrict the consideration of implication relations to those situations most likely to produce economical simplification.

As has already been implied, a row-pair (i,j) is used as the first element in an implication chain only if 1) one of the two states is incompatible with all known states, and 2) only a single pair of states (p,q) is implied by (i,j). These pairs are detected as the pair compatibility process previously described is executed: as state j is considered for compatibility with state i, if i and j are conditionally compatible and imply only a single pair of states (implicants) p and q, j is marked. If i is not found compatible with any known state, then an attempt is made to build a single implication chain based on (i,j) implies (p,q).

Implication chains which may be used to find valid compatibility classes terminate in several ways. If the final implicant pair p and q are unconditionally compatible, then all pairs in the chain are compatible. If a pair already in the chain is the only implicant of the last pair (i.e., the chain closes on itself), then all pairs in the chain are compatible.

A chain building attempt fails if some chain implicant pair (p,q) has two or more implicants, if a pair of implied states are incompatible, or if the chain length exceeds some threshold. The last has experimentally been shown to be unimportant; the restricted implications considered cause almost all chains to be very short. A fourth type of chain failure closely resembles a closed chain: if one state but not both of an implicant pair has previously appeared in a chain, the chain fails.

If a chain is successfully completed, compatibility classes are calculated in reverse order, beginning with the last class added to the chain. Rechecking may be performed after this operation is completed for all implicant pairs. The advisability of rechecking here is highly problem dependent but usually yields little additional simplification--at a relatively high cost.

Flow Table Reorganization.

After the process described above has been completed for each original flow table row, the flow table must be reorganized to eliminate the rows with positive tags and to complete the updating of next state entries.

Each row is considered in turn until all rows have been processed. If row i has a positive tag the flow table portion consisting of zero or negatively tagged rows above the "known" part of the table (with the window of the known rows based on row i) is examined for unstable entries valued i. These next state entries are changed to the appropriate state number of the class containing row i. A search is then made to find a row j i with a zero or negative tag to "fill" the space occupied by the eliminated row i. If such a row is found, next state entries are changed to reflect the relocation of row j to position i. If no rows are available to fill state i, the flow table reorganization process is complete.

Figure 3 illustrates the flow table reorganization procedure. Notice that the second row,

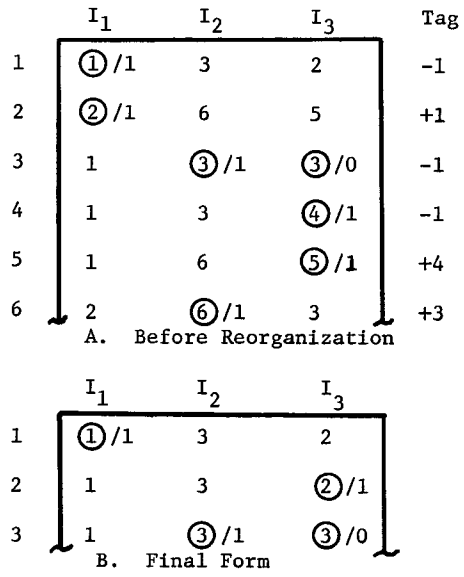


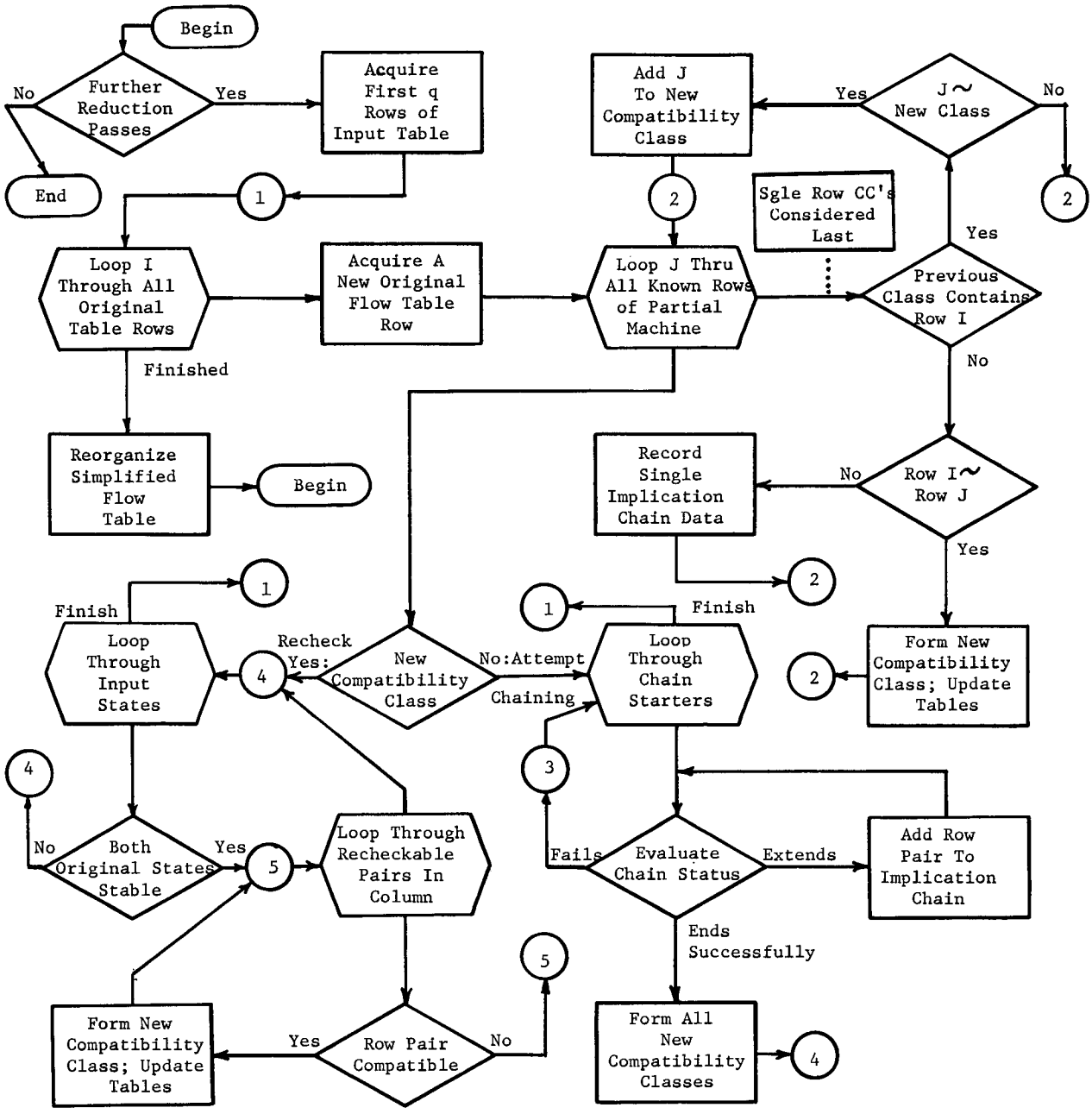
Figure 3. Reorganization of a Reduced Flow Table.

having a positive tag, is replaced by the lowest row 4 having a negative tag.

Algorithm Summary

The procedure outlined produces excellent simplification if a large portion of the flow table is in the range of consideration. However, the effort implied by such a large range is considerable. Thus, it is recommended that the procedure presented here be applied iteratively using a more economical range. Simplification processing ceases when a simplification yield requirement is not met.

Figure 4 is a brief flow diagram of the flow table simplification heuristic presented here.



Programmed Implementation and Results.

The flow table simplification heuristic described has been programmed in PL/1. Although the program will not be described in detail, the performance of the programmed procedure illustrates the utility of the simplification heuristic itself. It should be noted that the program was written in a high level language and emphasized algorithm clarity rather than execution efficiency.

Experience gained in several previously developed flow table simplification algorithms led to a program implementation of the procedure containing several minor modifications of the simplification heuristic described. These changes permitted the evaluation of constraints placed on various phases of the simplification process.

A rather trivial assumption was also made to allow an experimental simplification routine to be developed more rapidly. It was assumed that, as flow table simplification proceeds, enough memory is available to store all of the partially reduced machine. Thus as row i of the original flow table is added to the reduced machine, it is assumed that all compatibility classes containing rows 1 through i-1 are stored in main memory. This programming convenience eliminated the need for a partially reduced flow table segment paging and bookkeeping scheme--which although involving very significant extra programming effort is not technically important.

An interesting experimental modification of the program was a provision for varying the degree of "look ahead" used in the procedure. Although computer time costs have restricted experimentation with this parameter, some preliminary results can be reported.

The effect of various degrees of look-ahead are highly problem dependent, but processing times generally increase as look-ahead increases. The degree of reduction achieved, however, is usually less dependent on look-ahead, especially for values greater than 10%.

The examination of an extremely large number of single implication chains may be undesirable. The programmed simplification procedure thus contained a provision for halting the chain building process after a variable number of chain failures. A variable maximum chain length test was also incorporated (i.e., it failed all chains longer than the length limit). Both of these provisions were found to have almost no effect on either the degree of reduction obtained or execution time required. This result is due to the extremely low incidence of long single implication chains in the examples used, and the surprisingly small number of single implication chains discovered.

The programmed routine also contained an option for suppression of the iterated simplification feature. It was found that the increased simplification obtained was highly problem dependent. In some cases, virtually no simplification was achieved after the first pass. For other tables, significant reduction was obtained for up to three passes. In all cases, the amount of time required to complete a simplification cycle decreased markedly for successive passes.

The recheck performed after formation of new compatibility classes could also be bypassed in the programmed flow table simplification routine. It was found that rechecking contributes significantly to table reduction, especially on the first iteration.

on

Evaluation.

As has been pointed out previously, minimization procedures for the large flow tables considered here are impractical. Thus no attempt has been made to program a flow table minimization algorithm. Still, some means of evaluating the performance of the heuristic is essential.

The evaluation method selected consisted of constructing a completely simplified flow table by insuring that no two rows were compatible. A random number generator was then used to introduce redundant rows which had at least one stable state. Other next state entries in the redundant rows were randomly determined to be specified or unspecified. Finally, the rows of the redundant flow table were randomly reordered, and the table was prepared for simplification.

Several very large flow tables prepared in this manner have been reduced by the simplification heuristic program. Table I summarizes the results obtained using these examples and other large tables.

Test No.	No. Cols	Input Rows	Simp Rows	Min Rows	Time Secs
1	4	193	34	23	77
2	4	75	28	23	18
3	8	115	25	Unk	49
4	8	217	39	23	157
5	16	158	37	Unk	56

Table I. Simplification of Several Large Flow Tables.

The examples cited contain many rows and relatively few columns; such a flow table might be obtained from a list of desired input-output specifications.[5] Flow table simplification time using the algorithm described here is roughly proportional to the number of flow table cells (rows x columns) and hence is not dependent on flow table shape.

Conclusion.

The flow table reduction method described here represents a step toward the economical synthesis of large asynchronous sequential circuits. Performance of the programmed version of the simplification heuristic illustrates the utility of the method. Although the reduced tables are not minimal, considerable reduction is achieved at very low cost.

REFERENCES

1. Smith, Robert J. II, "Synthesis Heuristics for Large Asynchronous Sequential Circuits," Ph.D. Dissertation, Electrical Engineering Department, University of Missouri--Rolla, 1970.

2. Paull, M.C., and Unger, S.H., "Minimizing the Number of States in Incompletely Specified Sequential Switching Functions," IRETEC, September, 1959.

3. Grasselli, A., and Luccio, F., "A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks," IEEETEC, June, 1965.

4. Kella, J., "State Minimization of Incompletely Specified Sequential Machines," Unpublished Manuscript #R-68-107, IEEE Computer Group Repository, 1968.

5. Smith, R.J., and Tracey, J.H., "A New Method for Sequential Circuit Specification," Record of the 1970 SWIEEECO, April, 1970.

The research described in this paper was supported in part by NSF Grant GK 2017.