An Algebraic Technique for Generating Optimal CMOS Circuitry in Linear Time

L. S. NYLAND
Department of Computer Science, CB #3175
University of North Carolina, Chapel Hill, NC 27599-3175, U.S.A.
nyland@cs.unc.edu

J. H. REIF
Department of Computer Science, Duke University, Durham, NC 27708, U.S.A.
reif@cs.duke.edu

Abstract—We explore a method for quickly generating optimal CMOS functional circuits. The method is based upon an algebra we have derived that describes the composition of parallel-series graphs and their duals *simultaneously*, and as such, exactly describes the layout of CMOS functional circuits. The method is constructive; it creates the smallest components first, putting them together until the final circuit is realized. The constructed layout is representative of an unordered tree traversal, and is generated in time proportional to the number of input signals.

After describing the required concepts from graph theory and CMOS layout practices, we introduce an alternative symbolism for describing parallel-series graphs. We develop, with these symbols, a composition algebra, and demonstrate that the properties in the alternative domain hold in the original. We then use the algebra to implement a linear-time algorithm for generating CMOS functional cells.

Keywords—Boolean functions. Design automation, MOS integrated circuits, Integrated circuit layout, Optimization methods. Very Large Scale Integration (VLSI).

1. INTRODUCTION

In any VLSI design, there are always combinational switching functions that must be realized in silicon. We are proposing an automatic method for producing any combinational switching function (a function whose outputs depend only on the circuit inputs [1] with combining operators AND and OR) for CMOS layout. The error-free layout we generate is optimal in most cases, where optimal is defined by sharing diffusion (eliminating intertransistor gaps) as much as possible. The algorithm we use is based on an algebra that describes the combination of parallel-series graphs and their duals simultaneously, and thus is a perfect match for CMOS functional circuits. The problem is then: given a Boolean expression, find a layout that is minimal in size by having the minimum number of intertransistor gaps.

Considerable work has been done to automate the layout process of combinational switching functions. In [2], a heuristic algorithm is described that can produce optimal circuits, but only when it produces circuits with no gaps. If gaps are produced by their algorithm, it is not

This work has been supported in part by the National Science Foundation under contract CCR-8696134, the Office of Naval Research under contract ONR-N00014-87-K-0310, and in part by the Microelectronics Center of North Carolina. Approved for public release: distribution is unlimited.
We wish to thank D. Kozen for his suggestion to use an algebraic approach to solve the problem. We would also like to acknowledge the Microelectronics Center of North Carolina for their support.

Preprint of paper appearing in Computers and Mathematics with Applications Vol 31, No. 1, Jan. 1996, pp.85-108.

clear whether the layout is optimal or not. In [3], an algorithm to produce optimal results is described, based on the existence of *d-Eulerian* paths through the parallel-series graph of the function. Their solution depends on a large number of finite state machines that are complex to implement. In [4,5], two different methods of producing optimal layout are described, but only for pulldown networks (only the N-transistor network).

The work we have completed produces an optimal layout of switching functions in full complementary CMOS with a linear time algorithm that is consistent with their original expression (defined in Section 6). We have based this work on the results in [3,6]. There are no limits on the number of inputs imposed by our algorithm; however, if a large number is used, the time delay of the switching function can grow large. Our solution is simple, especially when compared to the solution given in [3]. It is also provably correct. A program that uses the results has been implemented and used.

In this paper we begin by reviewing some key concepts from graph theory, switching theory, and CMOS design that are necessary for the remainder of the paper. In Section 3 we describe parallel-series graphs and the representative graphs developed in [3]. Section 4 describes how we derived the algebra to combine subcomponents of a circuit. Section 5 describes the algorithm that uses the algebra to generate the layout, which is followed by an analysis of the results and the complexity of the algorithm. Section 7 describes the implementation of a program that utilizes this work. And finally, we state our conclusions.

2. A SHORT INTRODUCTION TO GRAPH THEORY, SWITCHING FUNCTIONS, AND CMOS LAYOUT

This section provides the necessary definitions that are required as a basis for the remainder of the paper. The definitions cover the necessary graph theory, switching theory, and simple CMOS design techniques.

2.1. Definitions from Graph Theory

A *graph* is a collection of vertices (or nodes) and edges (or arcs), where each edge connects two vertices in a graph. More formally, an *undirected graph* $G = (V, E)$ consists of a set of *vertices* V of size n and a set of *edges* E of size m. Each edge is an unordered pair (v, w) of disjoint vertices v and w. Graphs are often represented as points and lines, where the points are the vertices, and the lines are the connecting edges. Edges may also be independently labeled, rather than being labeled by the vertices they connect.

A *path* joining v_1 and v_k in G is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that for each i, $1 \leq i < k$ there is an edge $(v_i, v_{i+1}) \in E$. A path that traverses all edges exactly once is called *Eulerian*.

Parallel-series graphs are defined recursively. The trivial (smallest) parallel-series graph is a graph with two nodes and a single edge between them. All other parallel-series graphs consist of two smaller parallel-series graphs combined in series or parallel. A parallel series graph has two *distinguished vertices*, usually labeled s and t. The smallest instance of a parallel-series graph has only two nodes, so each is a distinguished node. If two parallel-series graphs are combined in *series*, then one distinguished node from each is merged together, and the other unmerged distinguished nodes remain distinguished. In a *parallel combination*, each distinguished node from one parallel-series graph is merged with one of the distinguished nodes in the other, forming the new, larger distinguished nodes of the result. Figure 1 shows an example of parallel-series graphs, combining operations, and distinguished nodes.

Parallel-series graphs are a subset of *planar graphs* and as such, parallel-series graphs have all the properties of planar graphs, such as a *dual* representation. A graph is *planar* if it can be drawn on the plane with no crossing edges. A planar graph defines a set of *faces* F, where a face is the area defined by the edges surrounding it.

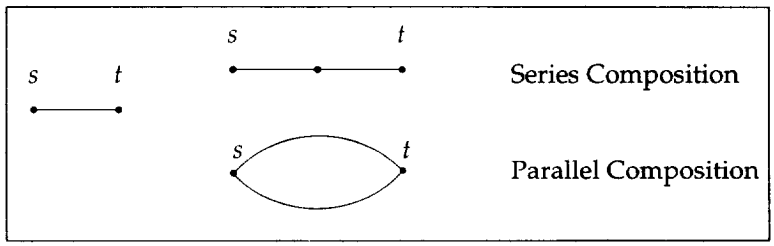


Figure 1. Simple parallel-series graphs, demonstrating series connections, parallel connections, and the distinguished nodes s and t.

The *dual of a planar embedded graph* G is the planar embedded graph $D(G)$ whose vertex set is F and the dual edges are the pairs (f, f') where there is an edge of E touching both f and f', and where these dual edges have a cyclic order consistent with the planar embedded graph of G.

There is an alternate way to describe the *dual of a parallel-series graph.* It is a parallel-series graph in which all the parallel connections are serial, and all the serial connections are parallel. The dual of a graph G is called $D(G)$. An example of a parallel-series graph and its dual is shown in Figure 2. The edges are labeled to show the opposite connection style between the graph and its dual.

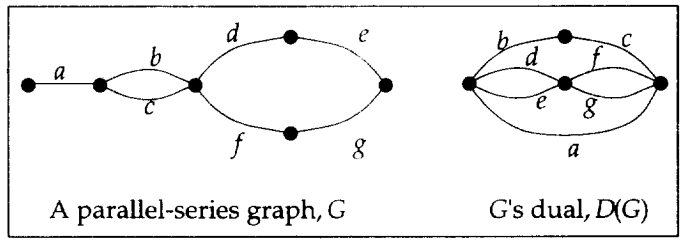


Figure 2. A parallel-series graph and its dual.

2.2. Switching Functions

The switching functions for which we are producing layout are multi-input, single-output functions and fall under the same assumptions for the layout of cells used in [2,3]. These are switching functions that consist of variables that are combined with AND/OR operators yielding a single output value based only on the input values. These cells are also referred to as *complex cells* or *functional cells.*

2.3. CMOS Layout

The CMOS layout of a cell consists of a row of P-transistors above a row of N-transistors in between power rails. Each P-transistor is above the N-transistor that is controlled by the same input signal, allowing the signal wires to run vertically forming the gate of each transistor. The AND/OR operations are realized by series and parallel connections among the transistors' sources and drains. The connections made on the P-transistors are the dual of the connections on the N-transistors [7]. An example layout of a NOR gate is shown in Figure 3(a).

This style of layout is widely accepted and is recommended by [8,9], because of the density of transistor placement, the sharing of wells around transistors of similar type, and protection against latch-up. When this layout style is used, adjacent transistors with electrically equivalent diffusion regions can be brought together to share a common diffusion region. Figure 3(b) shows an example of this. This style of layout also produces much denser layouts than networks of NAND and NOR gates.

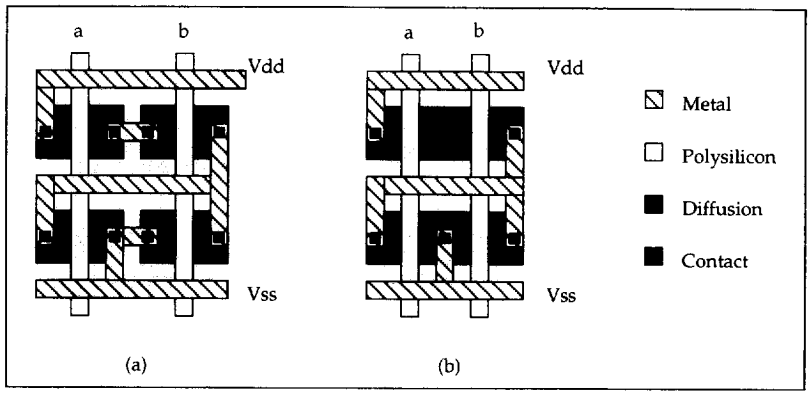


Figure 3. CMOS layout of a NOR gate.

We use a parallel-series graph G to represent the connectivity of the N-transistors, and the dual of G, $D(G)$ to represent the P-transistors. The transistors are represented by the edges in the graphs, while the electrically equivalent sources and drains of the transistors are represented by the nodes. Figure 3 shows the representation of an OR function of 2 inputs, along with representations of G and $D(G)$. One pair of distinguished vertices represent the power connections (G_s is the V_{ss} connection, and $D_s(G)$ is the V_{dd} connections), while the other pair $(G_t, D_t(G))$ are connected to form the output signal.

The output of the circuits generated produce the negation of Boolean function (this is the nature of CMOS layout). While there may be many different permutations of one Boolean expression that are all equivalent, different permutations can generate different layouts that may vary in size due to the number of gaps that must be introduced between transistors.

3. GRAPHS, REPRESENTATIVE GRAPHS

This section of the paper describes parallel-series graphs and the representative graphs developed in [3]. It also describes how both types of graphs are combined and then shows that the two are equivalent.

3.1. Parallel-Series Graphs

As [2,3] have previously shown, the type of switching functions being considered can be represented with parallel-series graphs. The edges in the graph represent the gates of transistors and the vertices represent electrically equivalent regions of the transistors' sources and drains. See Figure 4.

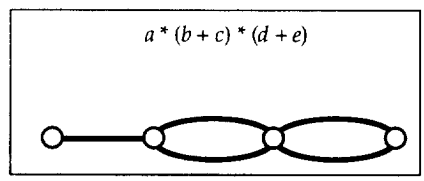


Figure 4. Parallel-series graph.

The connections on each row of transistors (P and N) can be represented by a parallel-series graph. We will refer to the graphs that represent the P-transistor connections as $G_p = (V_p, E_p)$ and the N-transistor connections as $G_n = (V_n, E_n)$. The edges E_p and E_n represent the gates of the transistors, and the vertices V_p and V_n represent the electrically equivalent regions of diffusion in the layout. The size of the sets of edges, E_p and E_n, are equal, since each edge represents a

transistor and there are the same number of P- and N-transistors. The sizes of the vertex sets, however, are not necessarily the same.

The dual of a graph is needed to describe the connections on the complementary set of transistors. Given one parallel-series graph that describes the connections of one type of transistor, the connections on the opposite type of transistors is merely its dual [7]. In our case, we use the parallel-series graph to represent the connections of the N-transistors, and its dual to represent the connections of the P-transistors.

A switching function is realized by series and parallel connections among individual transistors. To compute the negation of the AND operation over two signals controlling transistors, the P-transistors are connected in parallel, while the N-transistors are connected in series. For the OR operation, the P-transistors are connected in series and the N-transistors are connected in parallel.

Previous work [2.3] has shown the need to find Eulerian paths through both graphs to find the optimal layout. Recall that an Eulerian path is a path through a graph that traverses each edge exactly once. If one path is an Eulerian path in the graph and its dual, it is called *d-Eulerian* by [3]. If a d-Eulerian path is found, it dictates the order in which the transistors can be placed, sharing all diffusion regions. If a d-Eulerian path cannot be found, then the minimum set of paths that traverse both graphs is sought. The layout will have one less gap than the number of paths that traverse both graphs.

The problem is to minimize the size of the layout. This is done by finding the minimum set of paths that traverse both graphs simultaneously. We are not trying to minimize the number of transistors, that number is dictated by the switching function (one N- and one P-transistor for each variable). We are trying to find the smallest set of paths that traverse the graphs which is the same as minimizing the number of gaps in the layout. This also maximizes the number of shared diffusion regions between adjacent transistors.

In attempting to find the smallest set of paths that traverse both graphs, one must avoid the manipulation of a parallel-series graph into equivalent graphs. The manipulation consists of exchanging any two subgraphs that are in series with one another. While this may change the graph, the circuit it represents remains the same. When considering a parallel-series graph and its dual, there are at least $|E|/2$ subgraphs in series in the graph or its dual since the arcs in parallel in one graph are in series in the other. This manipulation is equivalent to evaluating different orderings of size n, where n is the number of subgraphs in series with one another. Since there are $n!$ different orderings of n elements, the exploration of all possible equivalent parallel-series graphs to achieve an optimal layout could be extremely time-consuming and is therefore unacceptable.

3.2. Representative Graphs

A better notation than parallel-series graphs is needed to describe the problem since parallel-series graphs contain more information than needed to solve the problem. Specifically, they contain the connectivity information about every vertex in the graph at all times. If we look at the problem as one of synthesizing the parallel-series graphs from smaller subgraphs instead of analyzing the complete parallel-series graphs, unnecessary information can be eliminated along the way. What is needed at each step is the connectivity information about the distinguished vertices (s,t) and some information about internal paths. Once a vertex becomes internal in a parallel-series graph (not a distinguished vertex), then very little information about that vertex is needed. It will not be directly used again in building larger parallel-series graphs. All of the information about internal vertices cannot be disregarded, however, because some information about the availability of paths through the graph must be maintained.

In [3], an alternate representation of the parallel-series graphs and their duals was introduced called *representative graphs*. They describe certain properties of a graph and its dual simultaneously and eliminate the internal clutter of parallel-series graphs, keeping only the information

needed to connect parallel-series graphs and their duals. The properties of representative graphs from [3] are listed here for completeness.

A *representative graph*, $H = (V, E)$ with index γ, is defined to represent the graphs G_n and G_p with a complete set of paths P if it has the following properties:

1. $V \subseteq \{v_{ss}, v_{st}, v_{ts}, v_{tt}\}$.
2. For all $v \in V$, degree$(v) \leq 1$.
3. For all vertices $v_{xy} \in V$, $x \in \{s_p, t_p\}$, $y \in \{s_n, t_n\}$, there exists a path in G_n which is also a path in G_p. Further, this vertex represents paths that have a terminal at the distinguished vertex x of G_p and y in G_n.
4. For all edges $(v_{xy}, v_{x'y'}) \in E$, $x, x' \in \{s_p, t_p\}$, $y, y' \in \{s_n, t_n\}$, there exists a path in G_p which is also a path in G_n. Further, this edge represents paths that terminate at distinguished vertices in both graphs (x to x' in G_p and y to y' in G_n).
5. The number of paths in P (as represented by H) in G_n which are also valid paths in G_p is exactly $\gamma + 1$ (see footnote[1]).
6. Both edges (v_{ss}, v_{tt}) and (v_{st}, v_{ts}) do not exist in a representative graph.

The notation for a vertex in a representative graph has a pair of subscripts, where each of the subscripts is either s or t. These represent the distinguished vertices from G_p and G_n, respectively. The notation $v_{x_p y_n}$ will be written as v_{xy} where it is implied that x is a distinguished vertex in G_p and y is a distinguished vertex in G_n.

Representative graphs are shown as a circle with up to four vertices inside, possibly with edges between any existing vertices. Each of the four possible vertices has a specific location inside the circular boundary. Figure 5 shows the explicit locations of the four potential vertices.

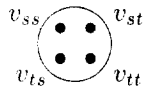


Figure 5. Positions of the 4 potential vertices in a representative graph.

It is important to note one fact that is not clearly elucidated in the properties listed. If there exists a vertex in a representative graph, then in the layout there exists one end of both rows of transistors that is a distinguished vertex. This is important since the solution to the layout problem involves placing sequences of transistors adjacent to one another. If it is known that there is (at least) one end of both rows of transistors on both subcircuits that is a distinguished vertex, the two subcircuits can be placed next to one another, overlapping the diffusion regions of the distinguished vertices, since both sides (P and N) have at least one connection to make. The additional arc needed for the parallel connection is simply added as a wire to be routed.

The value γ associated with every representative graph gives an indication of how many distinct paths there are in the complete set of paths for G_p and G_n. The larger the value of γ, the larger the layout will be. The way γ is computed is described in Section 3.3.

3.3. Combining Representative Graphs

Representative graphs can be combined with the AND and OR operators. To make effective use of the representative graphs, we must assume for the moment that the result of combining any two representative graphs is equivalent to the result of combining the two parallel-series graphs (and thereby the functional circuits) they represent. This section describes how two representative graphs are combined with the AND and OR operators, and then shows that the combination is an adequate representation.

[1]We have changed the meaning of gamma from [3] to what we consider a more meaningful indicator; that is, γ represents the number of gaps that need to be introduced in the layout.

For CMOS, the OR operator implies that the P networks are combined in series and the N networks are combined in parallel (see Figure 6). The AND operation is just the opposite; the P networks are combined in parallel and the N networks are combined in series.

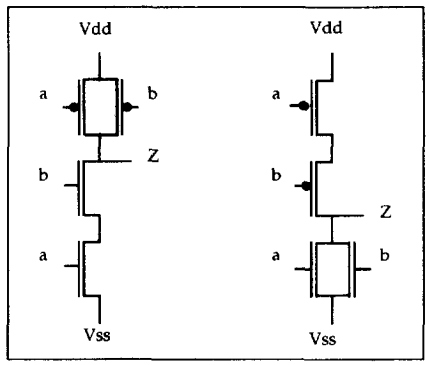


Figure 6. Two-input NAND and NOR circuits.

When two representative graphs are combined with the OR operator, one is simply placed above the other, some additional arcs are added, and then the two are reduced into one representative graph (see Figure 7). If the representative graphs are A and B, the additional arcs are (A_{ts}, B_{ss}) and (A_{tt}, B_{st}).

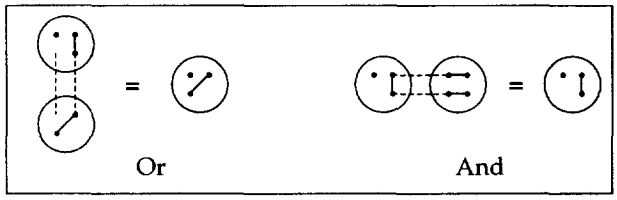


Figure 7. Combining representative graphs.

This connects the vertices

1. A_{t_p} with B_{s_p},
2. A_{s_n} with B_{s_n},
3. A_{t_p} with B_{s_p} (which is the same as the first),
4. A_{t_n} with B_{t_n}.

As the preceding list demonstrates, the P portion of the circuit is combined in series since there is only one connection (1 is the same as 3 in the above list) and the N portion of the circuit is combined in parallel since there are two distinct connections of distinguished vertices.

The reduction of two symbols into one for the completion of the OR operation consists of specifying the vertices and arcs for the resulting representative graph. The vertices A_{ss} and A_{st}, if they exist, make up the vertices in the top of the new graph, C_{ss} and C_{st}. The vertices B_{ts} and B_{tt}, if they exist, make up the vertices in the bottom of the new graph, C_{ts} and C_{tt}. Any paths that existed between these four vertices are paths in the new representative graph. See Figure 7.

The AND operation is identical to the OR operation, except the representative graphs A and B are placed side by side and the edges added are (A_{st}, B_{ss}) and (A_{tt}, B_{ts}). The vertices from the left side of A (A_{ss}, A_{ts}) and the right side of B (B_{st}, B_{tt}) make up the new set of vertices in the result.

The description of combining representative graphs in not yet complete. Two representative graphs may be combined in several different ways to implement either operation. Each representative graph can be rotated about the x- and y-axes. This yields between one and four different representative graphs for each graph, depending upon the amount of symmetry that exists in a given representative graph. Figure 8 shows an example of the four orientations of a representative graph with no symmetry.

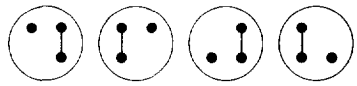


Figure 8. Possible orientations of representative graphs.

Combining two representative graphs, with four orientations each, may seem to yield as many as 16 different results. The results are all mirrored back to one form, immediately eliminating half of the 16 possibilities. Additionally, with most representative graphs, there is horizontal or vertical symmetry that reduces the number of distinct combinations down to a small number (consider the symbol with four vertices and no edges combined with itself, there is only one symbol generated). Figure 9 shows an example of all the symbols generated from two distinct representative graphs combined with the OR operator. The representative graph on the side of the table is placed above the representative graph along the top to produce the result in the matrix. When all generated symbols are translated back to one form, there are four distinct results.

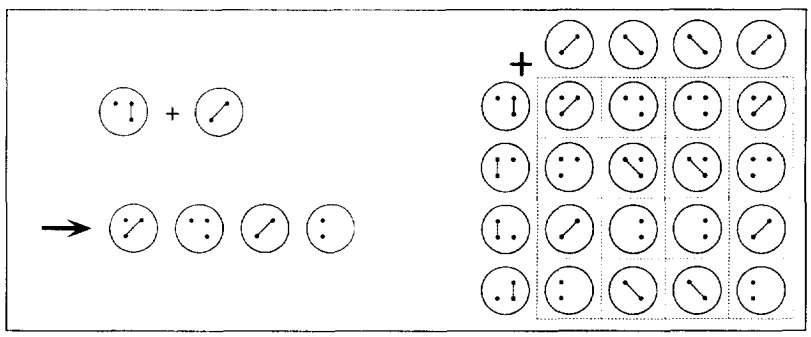


Figure 9. Results of combining two representative graphs.

Associated with each representative graph is an index, γ. It is the number of gaps that must be introduced in the layout of the representative graph. It is also one less than the number of paths that traverse the graphs (G_n and G_p). When two representative graphs, A and B, are combined, the resulting representative graph, C, must be assigned a new value of γ. It is computed by

$$C_\gamma = A_\gamma + B_\gamma + g,$$

where

$$g = \begin{cases} 0 & \text{if at least one of the added arcs between} \\ & A \text{ and } B \text{ has existing vertices on both ends,} \\ 1 & \textit{otherwise.} \end{cases}$$

This is not the same as the calculation for γ given in [3].

The values of g are explained as follows. If, during the combination, one of the new connecting arcs connects two existing vertices, then there is an orientation where distinguished vertices from the two subcircuits being combined can meet. If, however, neither added arc connects vertices from both representative graphs, then this orientation of the representative graphs does not reflect a layout with distinguished vertices that can be overlapped, and a gap must be introduced. The index γ must be increased in value by one to reflect this.

THEOREM 1. *The result of combining representative graphs represents the same combination of parallel-series graphs. That is, if*

$$\alpha \oplus \beta = \phi,$$

then

$$H_\alpha \oplus H_\beta = H_\phi,$$

where $\oplus$ *is either* $+$ *or* $*$, α, β, *and* ϕ *are parallel-series graphs, and* H_α, H_β, *and* H_ϕ *are the representative graphs for* α, β, *and* ϕ.

PROOF. The task of the proof is to show that H_ϕ adequately represents ϕ, which is the combination of α and β with either the OR or AND operator. We will show that by combining H_α with H_β to give H_ϕ by the method described that H_ϕ has all of the properties of a representative graph for the parallel-series graphs ϕ_n and its dual ϕ_p. The proof will be directed as if the operator used to combine α and β is $+$, but the argument for the $*$ operator is similar.

The first property of a representative graph states that the vertex set is a subset of $\{v_{ss}, v_{st}, v_{ts}, v_{tt}\}$. The vertex set of H_ϕ consists of the members v_{ss} and v_{st} from the vertex set of H_α and the members v_{ts} and v_{tt} from the vertex set of H_β. Therefore, the vertex set V of H_ϕ is a proper subset of the set listed in Property 1.

Property 2 states that no vertex has a degree greater than one. The vertices that exist in H_ϕ come directly from vertices that exist in H_α and H_β, each of which has a degree of 0 or 1. There are only two ways to produce edges in H_ϕ. The first is to copy the edge directly, and the second is to generate a new path from a longer path in the conjunction of H_α and H_β. In either case, the degree of the vertices remains the same, or in the cases where a path is incomplete, the degree of a vertex drops to zero. Therefore, if the degree of the vertices in H_α and H_β met the requirements of Property 2, then so do the vertices in H_ϕ.

Property 3 states that every vertex in H_ϕ represents a path in ϕ_n and ϕ_p and that the path represented has a terminal at one of the distinguished nodes in each parallel-series graph. The vertices in H_ϕ come directly from the vertices in H_α and H_β and represent the vertices not used in making the parallel-series connection. If a path existed in α and β that started at a distinguished node, and a connection was made with another graph leaving that node as a distinguished node, then the path that existed still exists as a path that has a terminal at a distinguished node. Thus, the existence of a vertex in H_ϕ represents a path in ϕ_n and ϕ_p with a terminal at a distinguished node.

Property 4 states that every edge in H_ϕ represents a path in ϕ_n and ϕ_p that has both terminals at distinguished nodes. An edge in H_ϕ originates in one of two ways: either it is a direct copy from one of the graphs H_α or H_β, or it is a new path derived from existing paths during the combination of H_α and H_β. See Figure 10 for an example of both edge derivations.

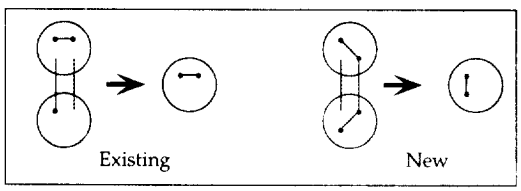


Figure 10. Origins of edges in representative graphs.

If the first case is true, then the path was in H_α or H_β and thereby met Property 4 before the combination. The distinguished nodes at the terminals of the path remain distinguished nodes after the combination, so the path still terminates on both ends at a distinguished node.

To generate a path in H_ϕ which is a composition of paths from H_α and H_β with the OR operator, there must have existed one path in each of H_α and H_β that had a vertical component which indicates that the path had terminals at opposite distinguished nodes in α_p and β_p. Furthermore,

α_p and β_p are the two graphs that will be connected in series, which means that the distinguished nodes used in the serial connection will no longer be distinguished. But if a path existed in each that spanned between the two distinguished nodes, then those two paths will be joined together as one path with terminals at the two remaining distinguished nodes. The parallel connection between α_n and β_n connects both distinguished nodes of one graph with distinct distinguished nodes of the other. If there is a path in each graph that has a terminal at the distinguished nodes being merged, then the two paths can be merged into one making a path that terminates on both ends in ϕ_n. Thus, a path in H_ϕ formed from one path in H_α and H_β has terminals at distinguished nodes of ϕ_n and ϕ_p.

There is an additional case that generates a new path in H_ϕ from paths in H_α and H_β. One example for the OR operation is the case where the edge set of H_α is $\{(v_{ss}, v_{ts}), (v_{st}, v_{tt})\}$ and the edge set of H_β has (v_{ss}, v_{st}) as a member. An example is shown in Figure 11.

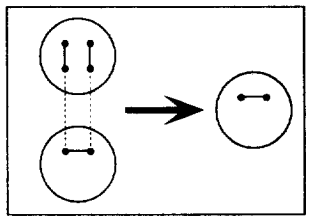


Figure 11. New edge created from 3 edges.

In all cases similar to this for the OR operation, there is one representative graph that has two vertical edges, and one representative graph with at least one horizontal edge. For the example given, there are two distinct paths between opposite distinguished nodes in α_p, and they map onto two distinct paths in α_n that start and end on the same distinguished node. See Figure 12.

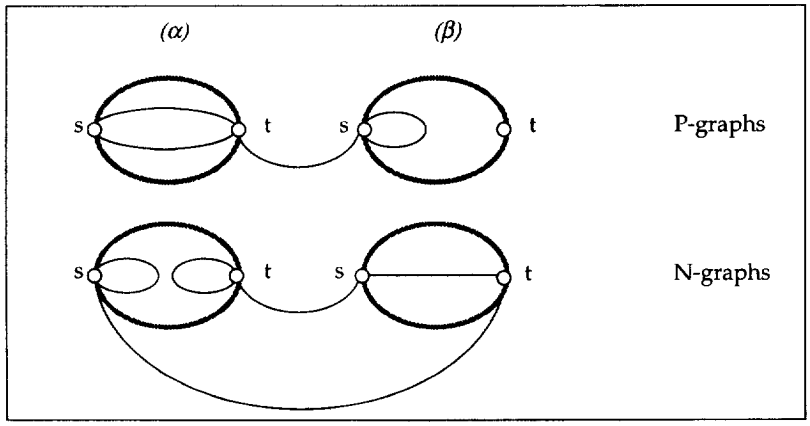


Figure 12. Merging of paths in N and P graphs.

If H_β has the only edge mentioned for the example listed, then there is a path that has terminals at the same distinguished node in β_p and a path with terminals at opposite distinguished nodes in β_n. The resulting paths that exist when the graphs are connected are one path in ϕ_p that starts and ends at the same distinguished node, and one path in ϕ_n which terminates at opposite distinguished nodes. Thus, there is one new edge in H_ϕ and it is (v_{ss}, v_{st}). For the OR operator, there are only four different cases that fit this description, and all produce the same result. We have now shown that Property 4 is true for H_ϕ.

Property 5 stipulates that the number of paths in P_ϕ is $\gamma_\phi + 1$. Let us examine the complete set of paths, P_ϕ.

Assuming that H_α and H_β are valid representative graphs, then there exist the complete sets of paths P_α and P_β. When the graphs are joined, it is clear that the union of P_α and P_β is a

complete set of paths for ϕ, so that there are no more than $|P_\alpha| + |P_\beta|$ elements in P_ϕ. However, there exists the possibility that one path from each of P_α and P_β can be joined into one path, and thereby set the number of members in P_ϕ to $|P_\alpha| + |P_\beta| - 1$.

The index γ represents the number of times that paths could not be joined, and is therefore one less than the number of paths in P. If the combination of graphs yields the extension of a path in P_α with a path from P_β, then the number of gaps (breaks in the paths) is simply the sum of the gaps in α and β. If the paths remain disjoint, then a gap is introduced, and the total number of gaps is increased by one. The computation is as follows:

$$|P_\alpha| = \gamma_\alpha + 1,$$
$$|P_\beta| = \gamma_\beta + 1.$$

If two paths are joined, then

$$|P_\phi| = |P_\alpha| + |P_\beta| - 1 = \gamma_\alpha + \gamma_\beta + 1,$$

implying

$$|P_\phi| = \gamma_\phi + 1$$

since

$$\gamma_\phi = \gamma_\alpha + \gamma_\beta.$$

Similarly, if there is no extension of paths between P_α and P_β, then

$$|P_\phi| = |P_\alpha| + |P_\beta| = \gamma_\alpha + \gamma_\beta + 2,$$

implying

$$|P_\phi| = \gamma_\phi + 1$$

since

$$\gamma_\phi = \gamma_\alpha + \gamma_\beta + 1.$$

Thus, the number of paths in P_ϕ that cover ϕ_n and ϕ_p is exactly $\gamma_\phi + 1$.

The last property to prove states that both diagonal edges (v_{ss}, v_{tt}) and (v_{st}, v_{ts}) do not exist in H_ϕ. The proof is by contradiction. Assume that both diagonal edges exist in H_ϕ and let us consider their origin from the OR operation. The construction of the edge (v_{ss}, v_{tt}) in H_ϕ has two possible origins: the edge (v_{ss}, v_{tt}) in H_α combined with the edge (v_{st}, v_{tt}) in H_β, or the edge (v_{ss}, v_{ts}) in H_α combined with the edge (v_{ss}, v_{tt}) in H_β. Similarly, to create the edge (v_{st}, v_{ts}) in H_ϕ there are also two possibilities: the edge (v_{st}, v_{ts}) in H_α combined with the edge (v_{ss}, v_{ts}) in H_β or the edge (v_{st}, v_{tt}) in H_α combined with the edge (v_{st}, v_{ts}) in H_β. Figure 13 shows both possibilities for the creation of both edges. There are two choices for (v_{ss}, v_{tt}) and two choices for (v_{st}, v_{ts}) giving a total of four possibilities. By examining all four possibilities that could possibly generate the edges (v_{ss}, v_{tt}) and (v_{st}, v_{ts}) in H_ϕ, either the degree of one of the vertices in H_α or H_β would have to be greater than one, or both diagonal edges would exist in either H_α or H_β. Neither of those possibilities can be true, since we know that H_α and H_β are valid representative graphs. Therefore, both diagonal edges cannot exist in H_ϕ.

Having shown that H_ϕ has all of the properties of a representative graph for the parallel-series graph ϕ_n and its dual ϕ_p, we have completed the proof of the theorem. ∎

The simplest parallel-series graph (that of a single arc) and its dual are described by the representative graph shown in Figure 14. It represents the simplest Boolean equation, that of a single variable, the result of which is a simple inverter implemented with one pair of complementary transistors. There is one path through each graph, which starts at s and ends at t in both graphs and has a γ of 0 (one path traverses both graphs entirely).

The 18 different representative graphs from [3] are shown in Appendix A. Ignoring different values of γ, all 18 of the symbols can be found by continually examining and saving results of OR and AND operations on all of the representative graphs, starting with the representative graph in Figure 14.

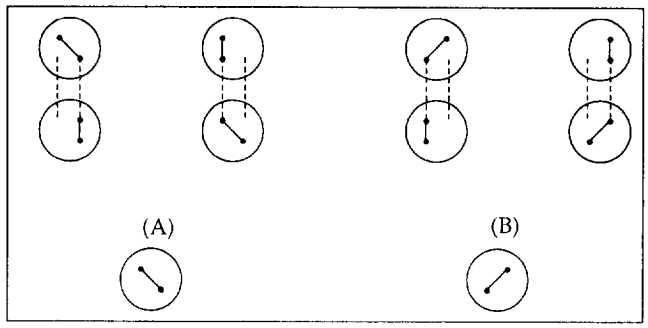


Figure 13. The 2 possible ways to create the 2 diagonal edges.

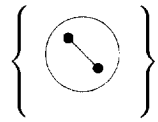
Figure 14. The initial representative graph.

4. ALGEBRA CREATION

In this section, we describe the creation of an algebra that denotes the combination of parallel-series graphs and their duals, using the representative graphs described in the previous section.

Recall that when two representative graphs are combined, the result may be more than one new representative graph due to the multiple orientations allowed. To build an algebra of symbols that will be capable of dealing with multiple representative graphs, every symbol in the algebra must denote a set of representative graphs. This means that instead of the starting symbol being

as mentioned in the previous section, it is actually



to indicate that the starting symbol of the algebra is a set with only one element.

LEMMA 1. *There is a finite number of algebraic symbols that can be found in a finite amount of time.*

PROOF. By examination, there are exactly 18 distinct representative graphs. The maximum number of combinations that must be examined is determined by pairwise combinations of all distinct subsets with both operators. The computation is

$$2^{18} * 2^{18} * 2 = 2^{37} \approx 10^{11}.$$

While this is a large number, it is an upper bound on the number of possible symbols in the algebra. ∎

The combination of one algebraic symbol with another (with the AND or OR operators) implies each representative graph in one symbol is combined with each representative graph in the other symbol. Recall that in the combination operation of two representative graphs, each representative graph can be mirrored in x and y. Intuitively, it may seem that a rapid growth in the number of representative graphs in each symbol will occur, but in actuality, the numbers remain small.

Rather than explore all possible combinations of subsets with both operators as outlined by Lemma 1, we use a constructive method to determine the algebra. The procedure consists of an

iteration over a *base* solution set of symbols, which initially contains only the starting symbol. During one iteration, each algebraic symbol is combined with every other symbol using the AND and OR operators, and the results of those operations are put into a new solution set. When all of the combinations have been computed, the base solution set is compared to the new solution set and if there is no difference, then the complete solution set has been found. If there is a difference, then the new solution set is made into the base solution set, and the process is repeated.

More precisely, the iterations continue as long as there exist two symbols a and b in the solution set (possibly the same symbol) where the result of $(a + b)$ or $(a * b)$ is not in the solution set.

When the process is finished, meaning that the combination of any two symbols with either operator yields a symbol already found, 36 symbols are enumerated using all 18 representative graphs. Additionally, tables for combining every symbol with every other symbol using either operator (AND/OR) are generated. See Appendices A and B for the actual values.

We have developed a specialized algebra that describes all possible AND/OR combinations of graphs and their duals simultaneously. It is an algebra with some, but not all, of the standard algebraic properties of a ring. A nonempty set R is said to be an *associative ring* (see [10]) if in R two operations are defined, denoted by $+$ and $*$, respectively, such that for all a, b, c in R:

1. $a + b$ is in R.
2. $a + b = b + a$.
3. $(a + b) + c = a + (b + c)$.
4. There is an element 0 in R such that $a + 0 = a$ (for every a in R).
5. There exists an element $-a$ in R such that $a + (-a) = 0$.
6. $a * b$ is in R.
7. $a * (b * c) = (a * b) * c$.
8. $a * (b + c) = a * b + a * c$ and $(b + c) * a = b * a + c * a$ (the two distributive laws).

Additionally, if the multiplication of R is such that $a * b = b * a$ for every a, b in R, then R is called a *commutative ring*. Finally, if a ring fails to have Property 7, it is called a nonassociative ring.

Let us examine which properties hold in our algebra, where $+$ and $*$ indicate the OR and AND operations.

- Closure (Properties 1 and 6) holds due to the fact that we sought and found a finite R where every a and b, when combined with either operator, yields an element of R.
- Commutativity (Property 2 and the definition of a commutative ring) is true since the order of the elements being combined does not matter. The reason order is not important is that the reorientation of the representative graphs during the combination operation is effectively the same as commuting the representative graphs. By allowing all of the reorientations of the representative graphs, we have commutativity.
- A disproof of associativity of the $+$ operator (Property 3) is the expression $(a + \overline{b}) + \overline{c} \neq a + (\overline{b} + \overline{c})$ where a, $\overline{b}$, and $\overline{c}$ are symbols in the algebra, not variables in a Boolean equation. The left side of the equation equals the symbol $\overline{j}$, whereas the right side equals $\overline{i}$.
- Similarly, associativity of the $*$ operator (Property 7) is falsified by $(a * b) * c \neq a * (b * c)$. The left side of the equation equals the symbol j, whereas the right side equals i.
- If Property 4 were true, there would exist a symbol x such that $a + x = a$. However, the symbol a (the starting symbol) never appears anywhere in the table itself. It is used only as an index into the table.
- Knowing that there is no element 0, based on the falseness of Property 4, Property 5 is not true because it implies that the element 0 exists.
- Distribution (Property 8) is not true since it changes the number of terms in the equations, thereby changing the number of transistors in a circuit.

Summarizing, we have derived an algebra that is commutative but not associative and is not distributive.

We have chosen to ignore one aspect of the problem which is maintaining the order of variables between the equation and the layout. We are seeking the optimal layout, which means a reordering of the variables might be necessary. However, if the vertical mirroring in the OR operation and the horizontal mirroring in the AND operation are not allowed during the combination of representative graphs, the order of the variables can be maintained.

This yields a substantial change in the number of algebraic symbols (84 of them, instead of 36), the tables describing the combinations of symbols, and the properties of the algebra. The commutativity is lost at the gain of associativity. We considered this approach briefly, but chose not to explore it because of the loss of optimality.

5. LAYOUT ALGORITHM

The findings from the previous section allow us to create an algorithm that produces optimal complementary layout of switching functions. The input to the algorithm is a Boolean switching function, and its output is the layout of a circuit that computes the negation of the Boolean function. This section describes this layout algorithm.

The expression whose layout is desired is available as input and is converted into a binary expression tree structure that dictates a relative ordering of each of the operators. This is a well-known technique used in many applications that deal with binary operators with a precedence ordering [11]. Figure 15 shows an example of an equation and its associated expression tree.

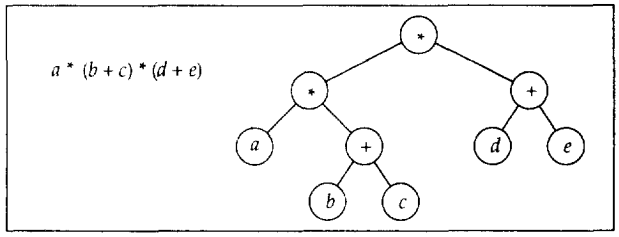


Figure 15. Binary expression tree.

Given a binary expression tree, the layout process can commence. It consists of three major phases: first, assign every node in the tree a symbol from the algebra; second, determine the proper ordering of the children for each node; and third, place and route the transistors based on the information stored in the expression tree. A description of each phase follows.

Prior to the first phase, each leaf in the expression tree is associated with one set of complementary transistors (one P- and one N-type) controlled by one input signal. Each leaf has also been assigned the starting symbol in the algebra, which we have labeled a. The internal nodes of the expression tree are associated with the operator that is needed to combine their children.

The goal of the first phase is to assign an algebraic symbol to every internal node in the tree. The symbols are found by looking up the result of combining the two symbols from the children in the tables derived in the previous section. The recursive procedure of assigning an algebraic symbol to each node in the tree involves: finding the symbol for the left child, finding the symbol for the right child, then using the symbols of each child and the operator combining them to assign a new symbol to the parent node. The recursion "bottoms out" at the leaves, since each leaf was originally assigned a symbol. The process is started at the root and the recursive procedure assigns a symbol to every node. When the root has been assigned a symbol, phase one is complete.

Figure 16 shows the symbols assigned to the expression tree for the equation $a*(b+c)*(d+e)$. At each node is the name of the signal or the operator enclosed in [ ] symbols. The name of the algebraic symbol and the representative graphs in that symbol are also shown at each node.

The name is to the left of the graphs and is followed by a ":". On the lower right side of each representative graph is its value of γ as computed for this expression.

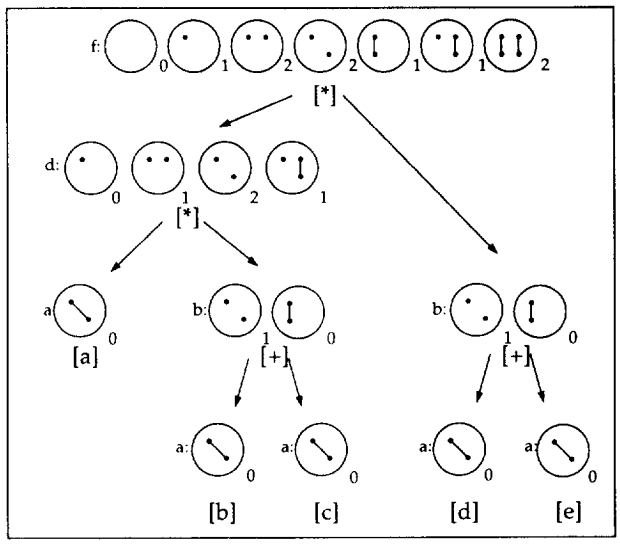


Figure 16. Assigning of algebraic symbols to nodes in the tree.

There are some additional details of the first phase that need to be pointed out. The lineage of each representative graph in each algebraic symbol is saved. This information is needed during the layout phase so that the descendants of any representative graph are known. For later use during the layout of the cell, pointers are kept to the two representative graphs (one from each child) that generated each representative graph in the parent node's symbol. If there is more than one way to generate a representative graph in the parent node's symbol, pointers are only kept to the two representative graphs in the children that generate the representative graph in the parent node with the lowest value of γ.

Each representative graph in the algebraic symbol at the root represents one possible layout and implies a choice of exactly one representative graph from each node below the root all the way down to the leaves. There may exist some representative graphs in the nodes below the root that are not referred to at all by representative graphs in the root's symbol, simply because when they were used to create a new representative graph, they produced the same symbol that two other representative graphs created but with a similar or larger value of γ.

The best representative graph to choose from the root's algebraic symbol is the one that produces the best layout, which is the one with the fewest gaps. This is indicated by the representative graph with the smallest value of γ. If there are several representative graphs with the same minimum value of γ, any one will do.

Figure 17 continues the example started in Figure 16. The best representative graph to choose from the root's symbol is ○ since it is the only representative graph where $\gamma = 0$. The arrow at the top of the figure indicates the chosen representative graph. Throughout the remainder of the tree, the representative graphs that are descendants of the symbol chosen at the root are emboldened. Naturally, every leaf contributed to the construction of the chosen representative graph at the root.

This brings us to the second phase of the layout process. The position of each leaf is not yet determined, since at no time have the locations of the distinguished nodes been specified for any of the children. Only the existence of the distinguished nodes is known by the algebra (or the representative graphs in the algebraic symbol). As an example, the representative graph ⊙ lets us know that there is only one end from both graphs, G_p and G_n, that is available for connection, and it is named s on both. The actual locations of the s node are not yet known and must be computed.

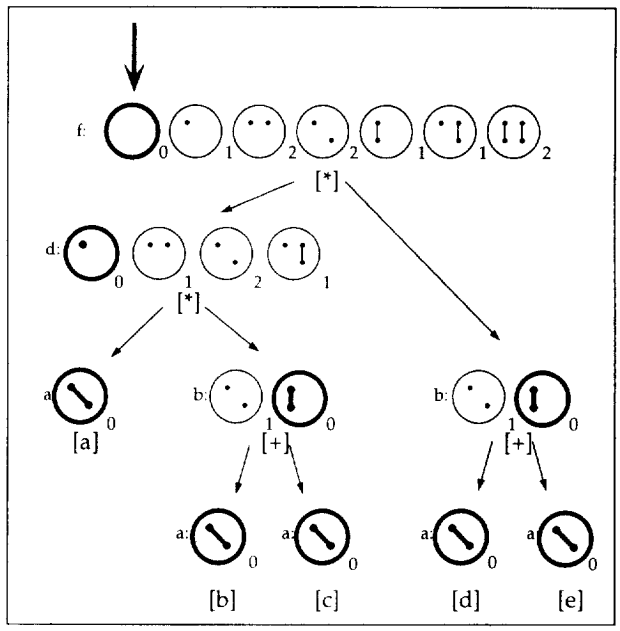


Figure 17. Choosing one representative graph in the root defines which representative graph to use from each node.

The goal of this phase is to place the available distinguished node on the correct end on the circuit layout so that proper abutment of subcircuits takes place. This is only a concern when the two subcircuits abut directly without a gap. If there is a gap between the children of a vertex, there is no orientation of the two subcircuits where distinguished nodes from both can overlap. This step is meaningless for a subcircuit with a gap between the left and right subcomponents.

We start by assuming that the children are in the proper order and proceed with a cursory examination of the tree to find those that are not. One assumption throughout the entire layout procedure is that left children will be placed to the left of right children. Wherever there is an attempt to layout a left child of a node that does not have a distinguished vertex on both rows of transistors at the right end, the left child must be flipped over since it is known that one end of the subcircuit is a distinguished vertex. The order of the left child's children is reversed and the same check is again made on the subtree rooted at the left child. The same reversal is done for right children without distinguished vertices on the left end.

This is not a local operation that can be done during the layout stage. The reordering of the children of a node may be done at any level in the tree. It is a post-order traversal of the tree which means that the order of the leaves is not determined until the order of the root's children is known. Therefore it must be done as a separate step, prior to the layout step.

The entire tree is passed over in a cursory examination stage, determining which subtrees must be exchanged. The result of this phase is a proper ordering of the leaves that will allow the layout specified by the representative graphs.

The third phase produces the actual layout. As with the previous phases, the layout is done in a recursive manner, which can be described as producing the layout for the left child, followed by the layout for the right child, and then completing the needed connections for this node in the subtree. The information needed to lay out a node is: which representative graph in the algebraic symbol is being used for layout and where the starting position for this node is. The information returned to this node's parent is the width of this subtree's layout and the sets of physical locations of the distinguished vertices, s and t, in the layout represented by G_n and G_p.

Between the steps of producing layout for the left and right children, it must be decided if the right child is laid out adjacent to or spaced away from the left child's layout. If $\gamma_{\text{parent}} > \gamma_{\text{left}} + \gamma_{\text{right}}$, then there is a gap placed between the left and right child. If $\gamma_{\text{parent}} = \gamma_{\text{left}} + \gamma_{\text{right}}$,

then there are distinguished vertices on the adjoining ends of the two subcomponents, which indicates that the layout of the two children can share diffusion regions. This information is then passed along for producing the layout of the right child in the correct location.

After the layout of the children is produced, the necessary connections between them must be made. Whether or not a gap has been introduced, the minimum length wires are sought for the connections between the two children. The connections between the left child A and the right child B using the AND operator are as follows.

One on the N-side for the series connection:

1. $\max(\max(A_{s_n}), \max(A_{t_n}))$ with the $\min(\min(B_{s_n}), \min(B_{t_n}))$.

And two on the P-side for the parallel connection:

1. $\max(\max(A_{s_p}), \max(A_{t_p}))$ with the $\min(\min(B_{s_p}), \min(B_{t_p}))$.
2. $\min(\max(A_{s_p}), \max(A_{t_p}))$ with the $\max(\min(B_{s_p}), \min(B_{t_p}))$.

It should be noted that A_{s_n} (as well as A_{s_p}, A_{t_n}, and A_{t_p}) represents a set of locations of a given electrical node at this point in the process. Thus, $\max(A_{s_n})$ means the largest x-coordinate of this electrical node. Connection 1 in each list merely states that the two distinguished vertices nearest each other should be connected, and if they are in the same position, no wire need be generated. Item 2 listed for the parallel connection merely states that those distinguished nodes not used in the first connection are to be used in the alternate parallel connection, but as short as possible.

We use the maximum and minimum positions to make the shortest possible connections. It is not necessary to pay strict attention to the connections of specific distinguished nodes because the names of the nodes are merely names and are free to be changed.

If the coordinates of the points to be connected are identical, then the connection is made by overlapping diffusion regions and no further work is needed to make the connection. If the coordinates of the connection are not identical, then the connection is put into a queue of connections to be made later by a routing step.

The connections for the OR operator are similar, only the p and n subscripts are exchanged.

6. ANALYSIS

The description of the algorithm is complete, so now we turn our attention to analyzing the size and optimality of the generated layout and the time taken to generate the layout. We feel that both aspects of our algorithm are quite good. First let us examine the size of the generated layout.

The actual problem we have solved is finding a global optimum of subcomponent conjunctions which is done by finding the minimal number of paths that traverse the graphs via the representative graphs. This method only finds paths that are consistent with the expression tree. A *consistent path* is a path through the graphs, G_n and G_p, that reflects an unordered traversal of the corresponding expression tree. An unordered tree-traversal is one where any order of traversing the children of a node is acceptable, but a node is only traversed when all children are traversed.

THEOREM 2. *For any parallel-series graph G_n and its dual G_p, there exists a set of representative graphs that represent all of the consistent paths through G_n and G_p and this set is exactly the set generated by the algebra.*

PROOF. The proof is by contradiction. Assume there exists a parallel-series graph G_n and its dual G_p that are not represented by an algebraic symbol (a set of representative graphs). Recall that all parallel-series graphs are originally constructed from graphs containing a single arc between the two distinguished nodes and that the original graphs are represented by the starting

symbol in the algebra. Any parallel-series graph can be described by a sequence of parallel-series combinations starting with the individual single-arc graphs. If there is a parallel-series graph that is not represented by a symbol in our algebra, then at some point there exist two subgraphs that are associated with an algebraic symbol such that when they are combined, they produce a set of representative graphs not represented by an algebraic symbol. But we know that every combination of two sets of representative graphs yield another set of representative graphs from Lemma 1. Therefore, all parallel-series graphs are represented by a set of representative graphs. ∎

THEOREM 3. *The layout generated for the given expression tree using the described algorithm is optimal over the set of consistent paths.*

PROOF. The optimal layout is the layout with the least number of gaps introduced between adjacent transistors. This is indicated by the representative graphs with the lowest value of γ in the set of representative graphs that represent the specified layout.

At every node in the tree, there is a set of representative graphs that represent the conjunction of the node's children with some of the worse conjunctions discarded (the same representative graph was generated two different ways with two different values of γ and only the better one is saved). The representative graph with the lowest value of γ represents the best layout for this node; however, the optimum for an interior node may not constitute part of the optimum for the root node, so all the representative graphs are retained, as well as the path to their creation.

When the root symbol is evaluated, all possible abutments in all possible orientations have been examined and saved. It is merely a process of choosing the optimal layout and extracting the orderings and orientations of the subcomponents. ∎

When the root or any subroot is evaluated, a representative graph is chosen to describe the layout, which determines a layout for each subtree. While the representative graph for the root is the one with the lowest value of the index γ, this does not imply that the optimal solution for each subtree is chosen.

As an example, the expression $a*(b+c)*(d+e)$ can be laid out by choosing the representative graph ○ at the root, which produces a layout with no gaps. By choosing this particular representative graph to lay out the expression, no distinguished vertices are available for abutment (there are no vertices in the representative graph). If the expression has another term added, $a*(b+c)*(d+e)*f$, then a different representative graph may be chosen for the layout of the subexpression $a*(b+c)*(d+e)$.

We have shown the layout results are optimal with respect to our definition. It is equally important to examine the time complexity of the algorithm, for there are many optimal graph algorithms whose time complexity is too large to make any use of them.

THEOREM 4. *The time complexity of the algorithm is* $O(n)$.

PROOF. Recall the major steps outlined in Section 5. A binary tree with n leaves must be constructed; an algebraic symbol must be assigned to every node in the tree; the ordering of the children of each node of the tree must be determined; and finally, the layout given by the tree must be generated.

The number of leaves in the tree is simply the number of P/N transistor pairs in the layout, which is also the number of variables in the input expression. This is the metric, n, that describes the problem size.

In a binary tree with n leaves, there are exactly $2n-1$ nodes in the tree. Each major phase outlined above traverses the tree once, with an equivalent amount of work being performed at each node in each step. Some of the steps do not involve any work at the leaves (there are $n-1$ internal nodes in a binary tree).

Every major step in the algorithm takes time $C_i * (2n - 1)$ or $C_i * (n - 1)$, where C_i is a computation constant for the i^{th} step. The algorithm consists of three phases, none of which has a time complexity larger than $O(n)$. Thus, the execution time of the algorithm is directly proportional to the number of inputs. ∎

7. IMPLEMENTATION NOTES

A program that produces the algebra described in Section 4 has been implemented, and the data it produces is used by a second program that generates layout of Boolean switching functions. Both programs are written in C and run on several different processors in several environments. In actual runs that have been monitored, the execution time of the layout generator is dominated by I/O operations rather than the actual computation time.

A straight-line router was implemented to make the N-transistor and P-transistor connections. The router is very simple. The size of the transistors chosen gives the router three horizontal tracks in which to do the routing. The connections for V_{dd}, V_{ss} and the output signal are made first to the tracks on the transistors to which they are closest. After that, each connection request is routed in a straight line. An attempt to find a previously completed route of the same electrical node is done first to eliminate the addition of unnecessary contact cuts. If one is not found, then a clear track from one end of the connection to the other is sought and used for the routing track. If a path is not found, a *pseudo*-wire is inserted for another program to route later (this provision has not been used in any runs to date).

The input to the layout program is a binary expression consisting of signal names (variables) with the operators + and * for the OR and AND operators, respectively. Parentheses are also allowed to specify orderings of the operators.

The program to generate the layout performs within the VIVID environment developed at the Microelectronics Center of North Carolina (MCNC). The output is a cell in the ABCD language [12], which is a symbolic circuit layout language with topological information. It contains the placement information of all transistors and wires as well as signal names controlling the gates of the transistors. The generated cell can be used with the VIVID compactor, HCOMPACT [13], to generate physical layout from the symbolic layout for any one of a number of CMOS fabrication lines (MOSIS, MCNC, etc.). The cell can be used at the symbolic level with other cells, either automatically or hand-generated, or at the mask level with other mask-level cells.

8. CONCLUSIONS

We have found a method of generating optimal layout of combinational switching functions from a Boolean expression with a time complexity proportional to the number of variables in the expression. Additionally, we have derived an algebra that describes all combinations of parallel-series graphs and their duals simultaneously.

Using the algebra and layout algorithm developed here, we have eliminated the task of producing layout for switching functions by hand. This includes the tasks of layout, verifying design-rule correctness, and connectivity verification.

Since the procedure is fast, the layout of any Boolean expression can be quickly generated, and it is known that the results are optimal.

One aspect of the problem that has not been explored is the rearrangement of the binary expression trees prior to layout. This may prove to be an important step considering that the algebra developed is not associative. Further work is being done is this area.

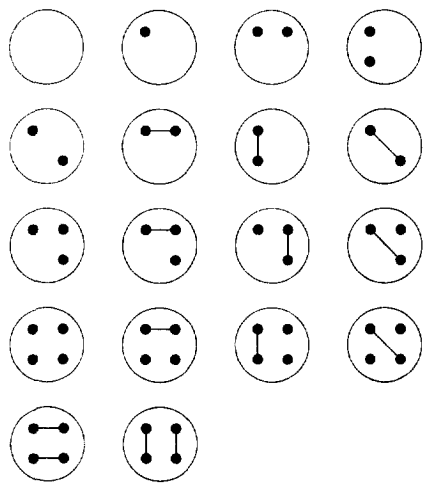
Another open problem to consider has to do with the restriction of using parallel-series graphs. While the intrinsic structure of parallel-series graphs was the basis for much of our work, it would be a great leap forward to be able to layout a circuit represented by an arbitrary planar graph optimally.

REFERENCES

1. Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, (1978).
2. T. Uehara and W.M. van Cleemput, Optimal layout of CMOS functional arrays, *IEEE Transactions on Computers* **C-30** (5), 305–312 (May 1981).
3. R. Nair, A. Bruss and J. Reif, Linear time algorithms for optimal CMOS layout, In *VLSI: Algorithms and Architectures, Proceedings of the International Workshop on Parallel Computing and VLSI*, May 23–25, 1985, (Edited by P. Bértolazzi and F. Luccio), pp. 327–338, North-Holland.
4. R. Müller and T. Lengauer, Linear algorithms for two CMOS layout problems, In *Proc. of Aegean Workshop on Computing VLSI Algorithms and Architectures*, (Edited by F. Makedon *et al.*), Springer Lecture Notes in Computer Science, No. 227, pp. 121–132, (1985).
5. T. Lengauer and R. Müller, Linear algorithms for optimizing the layout of dynamic CMOS cells, Technical Report 32, University of Paderborn, Paderborn, Germany, (December 1986).
6. D. Kozen, Personal communications about algebraic foundations of the composition of parallel-series graphs and their duals.
7. S. Muroga, *VLSI System Design*, John Wiley & Sons, (1982).
8. N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, (1985).
9. J. Poulton, C.D. Rogers and V. Leary, *CMOS Design Style Reference Manual*, Microelectronics Center of North Carolina, (1984).
10. I.N. Herstein, *Topics in Algebra*, 2nd edition, Xerox College Publishing, (1975).
11. A.V. Aho and J.D. Ullman, *Principles of Compiler Construction*, Addison-Wesley, (1979).
12. J. Rosenberg, S. Daniel, N. Weste and C.D. Rogers, *ABCD—A Better Circuit Description Language*, Microelectronics Center of North Carolina, (1982).
13. G. Entenman and S. Daniel, *Hcompact—The VIVID Hierarchical Compactor*, (1984).
14. E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, (1976).

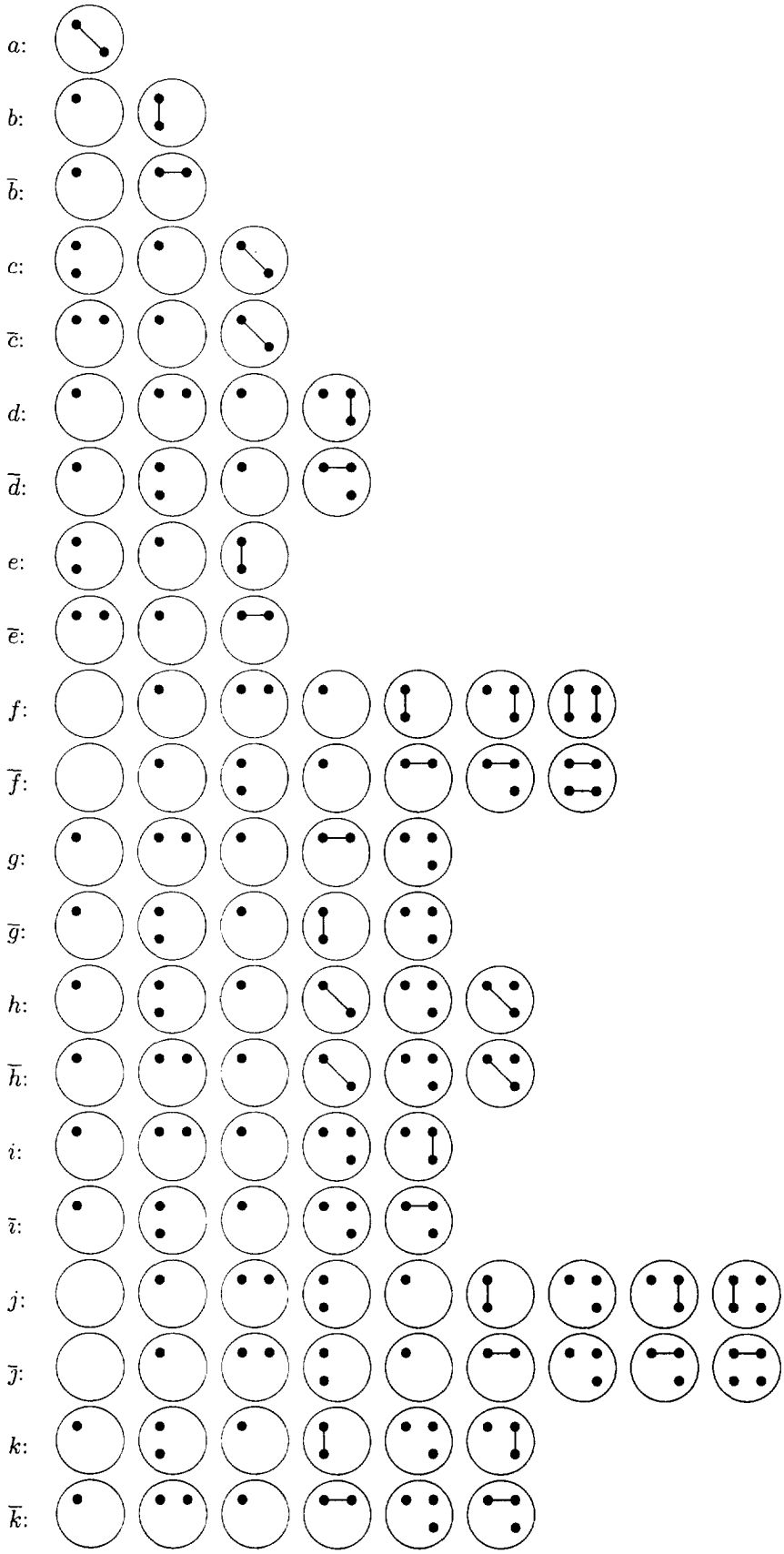
APPENDIX A
THE REPRESENTATIVE GRAPHS

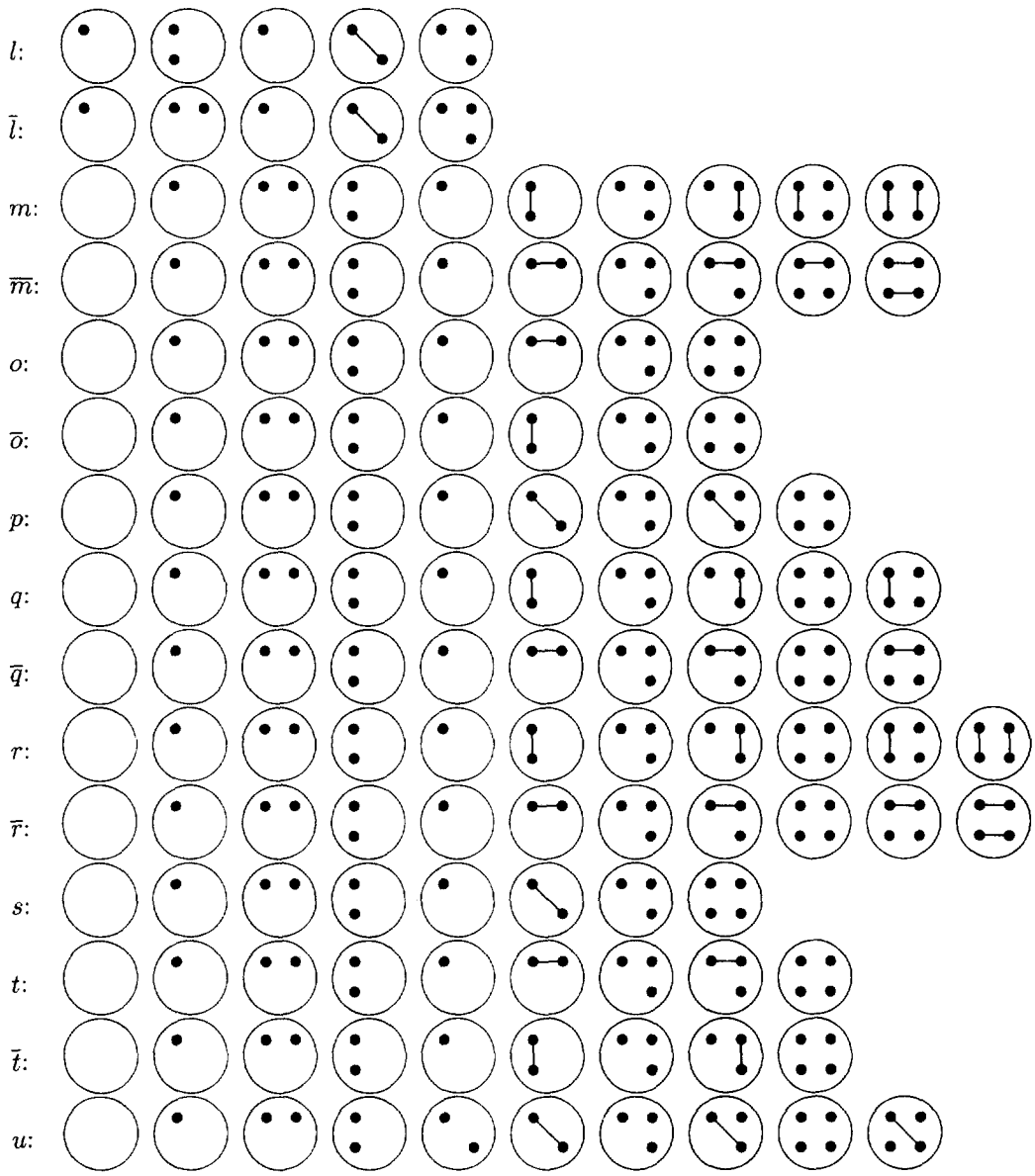
The following diagram shows the 18 *representative graphs* as they appear in [3]. The existence of these were all verified by the algebra derivation program.



The remainder of this appendix shows the representative graphs constituting symbols in the algebra found by our program. The solution set initially only contains the symbol a. Note equality of certain symbols and their opposite. The overbar indicates a complementary symbol, where the graphs G_n and G_p are swapped. Considering the two simple operations $a + a = b$ and $a * a = \overline{b}$, it can be seen that the paths in b and $\overline{b}$ are merely exchanged between the N and P regions. Some symbols are equivalent to their complement and are only listed once ($a = \overline{a}$, $p = \overline{p}$, $s = \overline{s}$, $u = \overline{u}$).

The Symbols of the Algebra




l:
$\bar{l}$:
m:
$\overline{m}$:
o:
$\bar{o}$:
p:
q:
$\bar{q}$:
r:
$\bar{r}$:
s:
t:
$\bar{t}$:
u:

APPENDIX B
TABLES

This appendix contains tables that show the result of combining any two algebraic symbols with either the + (OR) or * (AND) operator.

OR Table.

+	a	b	$\bar{b}$	c	$\bar{c}$	d	$\bar{d}$	e	$\bar{e}$	f	$\bar{f}$	g	$\bar{g}$	h	$\bar{h}$	i	$\bar{i}$	j	$\bar{j}$	k	$\bar{k}$	l	$\bar{l}$	m	$\bar{m}$	o	$\bar{o}$	p	q	$\bar{q}$	r	$\bar{r}$	s	t	$\bar{t}$	u
a	b	c	$\bar{d}$	e	$\bar{g}$	h	$\bar{d}$	c	$\bar{i}$	h	$\bar{d}$	$\bar{i}$	l	k	k	h	$\bar{i}$	h	$\bar{i}$	h	$\bar{i}$	$\bar{g}$	$\bar{g}$	h	$\bar{i}$	$\bar{i}$	l	k	h	$\bar{i}$	h	$\bar{i}$	$\bar{g}$	$\bar{i}$	h	k
b	c	e	$\bar{d}$	c	l	k	$\bar{d}$	e	$\bar{i}$	k	$\bar{d}$	$\bar{i}$	$\bar{g}$	h	h	k	$\bar{i}$	k	$\bar{i}$	k	$\bar{i}$	l	l	k	$\bar{i}$	$\bar{i}$	$\bar{g}$	h	k	$\bar{i}$	k	$\bar{i}$	l	$\bar{i}$	k	h
$\bar{b}$	$\bar{d}$	$\bar{d}$	$\bar{f}$	$\bar{d}$	$\bar{j}$	$\bar{j}$	$\bar{f}$	$\bar{d}$	$\bar{m}$	$\bar{j}$	$\bar{f}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{m}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{j}$
c	e	c	$\bar{d}$	e	$\bar{g}$	h	$\bar{d}$	c	$\bar{i}$	h	$\bar{d}$	$\bar{i}$	l	k	k	h	$\bar{i}$	h	$\bar{i}$	h	$\bar{i}$	$\bar{g}$	$\bar{g}$	h	$\bar{i}$	$\bar{i}$	l	k	h	$\bar{i}$	h	$\bar{i}$	$\bar{g}$	$\bar{i}$	h	k
$\bar{c}$	$\bar{g}$	l	$\bar{j}$	$\bar{g}$	$\bar{o}$	p	$\bar{j}$	l	$\bar{q}$	p	$\bar{j}$	$\bar{q}$	s	$\bar{t}$	$\bar{t}$	p	$\bar{q}$	p	$\bar{q}$	p	$\bar{q}$	s	$\bar{t}$	p	$\bar{q}$	p	$\bar{q}$	$\bar{o}$	$\bar{q}$	p	$\bar{t}$	$\bar{q}$	p	$\bar{q}$	$\bar{o}$	$\bar{q}$	p	$\bar{t}$
d	h	k	$\bar{j}$	h	p	q	$\bar{j}$	k	$\bar{q}$	q	$\bar{j}$	$\bar{q}$	$\bar{t}$	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	$\bar{q}$	$\bar{t}$	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u
$\bar{d}$	$\bar{d}$	$\bar{d}$	$\bar{f}$	$\bar{d}$	$\bar{j}$	$\bar{j}$	$\bar{f}$	$\bar{d}$	$\bar{m}$	$\bar{j}$	$\bar{f}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{m}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{j}$
e	c	e	$\bar{d}$	c	l	k	$\bar{d}$	e	$\bar{i}$	k	$\bar{d}$	$\bar{i}$	$\bar{g}$	h	h	k	$\bar{i}$	k	$\bar{i}$	k	$\bar{i}$	l	l	k	$\bar{i}$	$\bar{i}$	$\bar{g}$	h	k	$\bar{i}$	k	$\bar{i}$	l	$\bar{i}$	k	h
$\bar{e}$	$\bar{i}$	$\bar{i}$	$\bar{m}$	$\bar{i}$	$\bar{q}$	$\bar{q}$	$\bar{m}$	$\bar{i}$	$\bar{r}$	$\bar{q}$	$\bar{m}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$
f	h	k	$\bar{j}$	h	p	q	$\bar{j}$	k	$\bar{q}$	r	$\bar{m}$	$\bar{q}$	$\bar{t}$	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	r	$\bar{r}$	$\bar{q}$	$\bar{t}$	u	q	$\bar{q}$	r	$\bar{r}$	p	$\bar{q}$	q	u
$\bar{f}$	$\bar{d}$	$\bar{d}$	$\bar{f}$	$\bar{d}$	$\bar{j}$	$\bar{j}$	$\bar{f}$	$\bar{d}$	$\bar{m}$	$\bar{m}$	$\bar{f}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{m}$	$\bar{m}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{m}$	$\bar{m}$	$\bar{m}$	$\bar{j}$	$\bar{m}$	$\bar{j}$	$\bar{j}$	$\bar{j}$	$\bar{j}$
g	$\bar{i}$	$\bar{i}$	$\bar{m}$	$\bar{i}$	$\bar{q}$	$\bar{q}$	$\bar{m}$	$\bar{i}$	$\bar{r}$	$\bar{q}$	$\bar{m}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$
$\bar{g}$	l	$\bar{g}$	$\bar{j}$	l	s	$\bar{t}$	$\bar{j}$	$\bar{g}$	$\bar{q}$	$\bar{t}$	$\bar{j}$	$\bar{q}$	$\bar{o}$	p	p	$\bar{t}$	$\bar{q}$	$\bar{t}$	$\bar{q}$	s	s	$\bar{t}$	$\bar{q}$	$\bar{q}$	$\bar{o}$	p	$\bar{t}$	$\bar{q}$	$\bar{t}$	$\bar{q}$	s	$\bar{q}$	$\bar{t}$	p
h	k	h	$\bar{j}$	k	$\bar{t}$	u	$\bar{j}$	h	$\bar{q}$	u	$\bar{j}$	$\bar{q}$	p	q	q	u	$\bar{q}$	u	$\bar{q}$	$\bar{t}$	$\bar{t}$	u	$\bar{q}$	$\bar{q}$	p	q	u	$\bar{q}$	u	$\bar{q}$	$\bar{t}$	$\bar{q}$	u	q		
$\bar{h}$	k	h	$\bar{j}$	k	$\bar{t}$	u	$\bar{j}$	h	$\bar{q}$	u	$\bar{j}$	$\bar{q}$	p	q	q	u	$\bar{q}$	u	$\bar{q}$	$\bar{t}$	$\bar{t}$	u	$\bar{q}$	$\bar{q}$	p	q	u	$\bar{q}$	u	$\bar{q}$	$\bar{t}$	$\bar{q}$	u	q		
i	h	k	$\bar{j}$	h	p	q	$\bar{j}$	k	$\bar{q}$	q	$\bar{j}$	$\bar{q}$	$\bar{t}$	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	$\bar{q}$	$\bar{t}$	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u
$\bar{i}$	$\bar{i}$	$\bar{i}$	$\bar{m}$	$\bar{i}$	$\bar{q}$	$\bar{q}$	$\bar{m}$	$\bar{i}$	$\bar{r}$	$\bar{q}$	$\bar{m}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$
j	h	k	$\bar{j}$	h	p	q	$\bar{j}$	k	$\bar{q}$	q	$\bar{j}$	$\bar{q}$	$\bar{t}$	u	u	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	$\bar{q}$	$\bar{t}$	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u		
$\bar{j}$	$\bar{i}$	$\bar{i}$	$\bar{m}$	$\bar{i}$	$\bar{q}$	$\bar{q}$	$\bar{m}$	$\bar{i}$	$\bar{r}$	$\bar{q}$	$\bar{m}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$
k	h	k	$\bar{j}$	h	p	q	$\bar{j}$	k	$\bar{q}$	q	$\bar{j}$	$\bar{q}$	$\bar{t}$	u	u	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	$\bar{q}$	$\bar{t}$	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u		
$\bar{k}$	$\bar{i}$	$\bar{i}$	$\bar{m}$	$\bar{i}$	$\bar{q}$	$\bar{q}$	$\bar{m}$	$\bar{i}$	$\bar{r}$	$\bar{q}$	$\bar{m}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$
l	$\bar{g}$	l	$\bar{j}$	$\bar{g}$	$\bar{o}$	p	$\bar{j}$	l	$\bar{q}$	p	$\bar{j}$	$\bar{q}$	s	$\bar{t}$	$\bar{t}$	p	$\bar{q}$	p	$\bar{q}$	$\bar{o}$	$\bar{o}$	p	$\bar{q}$	$\bar{q}$	s	$\bar{t}$	p	$\bar{q}$	p	$\bar{q}$	$\bar{o}$	$\bar{q}$	p	$\bar{t}$		
$\bar{l}$	$\bar{g}$	l	$\bar{j}$	$\bar{g}$	$\bar{o}$	p	$\bar{j}$	l	$\bar{q}$	p	$\bar{j}$	$\bar{q}$	s	$\bar{t}$	$\bar{t}$	p	$\bar{q}$	p	$\bar{q}$	$\bar{o}$	$\bar{o}$	p	$\bar{q}$	$\bar{q}$	s	$\bar{t}$	p	$\bar{q}$	p	$\bar{q}$	$\bar{o}$	$\bar{q}$	p	$\bar{t}$		
m	h	k	$\bar{j}$	h	p	q	$\bar{j}$	k	$\bar{q}$	r	$\bar{m}$	$\bar{q}$	$\bar{t}$	u	u	q	$\bar{q}$	q	$\bar{q}$	p	p	r	$\bar{r}$	$\bar{q}$	$\bar{t}$	u	q	$\bar{q}$	r	$\bar{r}$	p	$\bar{q}$	q	u		
$\bar{m}$	$\bar{i}$	$\bar{i}$	$\bar{m}$	$\bar{i}$	$\bar{q}$	$\bar{q}$	$\bar{m}$	$\bar{i}$	$\bar{r}$	$\bar{r}$	$\bar{m}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$
o	$\bar{i}$	$\bar{i}$	$\bar{m}$	$\bar{i}$	$\bar{q}$	$\bar{q}$	$\bar{m}$	$\bar{i}$	$\bar{r}$	$\bar{q}$	$\bar{m}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$
$\bar{o}$	l	$\bar{g}$	$\bar{j}$	l	s	$\bar{t}$	$\bar{j}$	$\bar{g}$	$\bar{q}$	$\bar{t}$	$\bar{j}$	$\bar{q}$	$\bar{o}$	p	p	$\bar{t}$	$\bar{q}$	$\bar{t}$	$\bar{q}$	s	s	$\bar{t}$	$\bar{q}$	$\bar{q}$	$\bar{o}$	p	$\bar{t}$	$\bar{q}$	$\bar{t}$	$\bar{q}$	s	$\bar{q}$	$\bar{t}$	p		
p	k	h	$\bar{j}$	k	$\bar{t}$	u	$\bar{j}$	h	$\bar{q}$	u	$\bar{j}$	$\bar{q}$	p	q	q	u	$\bar{q}$	u	$\bar{q}$	$\bar{t}$	$\bar{t}$	u	$\bar{q}$	$\bar{q}$	p	q	u	$\bar{q}$	u	$\bar{q}$	$\bar{t}$	$\bar{q}$	u	q		
q	h	k	$\bar{j}$	h	p	q	$\bar{j}$	k	$\bar{q}$	q	$\bar{j}$	$\bar{q}$	$\bar{t}$	u	u	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	$\bar{q}$	$\bar{t}$	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u		
$\bar{q}$	$\bar{i}$	$\bar{i}$	$\bar{m}$	$\bar{i}$	$\bar{q}$	$\bar{q}$	$\bar{m}$	$\bar{i}$	$\bar{r}$	$\bar{q}$	$\bar{m}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$
r	h	k	$\bar{j}$	h	p	q	$\bar{j}$	k	$\bar{q}$	r	$\bar{m}$	$\bar{q}$	$\bar{t}$	u	u	q	$\bar{q}$	q	$\bar{q}$	p	p	r	$\bar{r}$	$\bar{q}$	$\bar{t}$	u	q	$\bar{q}$	r	$\bar{r}$	p	$\bar{q}$	q	u		
$\bar{r}$	$\bar{i}$	$\bar{i}$	$\bar{m}$	$\bar{i}$	$\bar{q}$	$\bar{q}$	$\bar{m}$	$\bar{i}$	$\bar{r}$	$\bar{r}$	$\bar{m}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$
s	$\bar{g}$	l	$\bar{j}$	$\bar{g}$	$\bar{o}$	p	$\bar{j}$	l	$\bar{q}$	p	$\bar{j}$	$\bar{q}$	s	$\bar{t}$	$\bar{t}$	p	$\bar{q}$	p	$\bar{q}$	$\bar{o}$	$\bar{o}$	p	$\bar{q}$	$\bar{q}$	s	$\bar{t}$	p	$\bar{q}$	p	$\bar{q}$	$\bar{o}$	$\bar{q}$	p	$\bar{t}$		
t	$\bar{i}$	$\bar{i}$	$\bar{m}$	$\bar{i}$	$\bar{q}$	$\bar{q}$	$\bar{m}$	$\bar{i}$	$\bar{r}$	$\bar{q}$	$\bar{m}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{r}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$	$\bar{q}$
$\bar{t}$	h	k	$\bar{j}$	h	p	q	$\bar{j}$	k	$\bar{q}$	q	$\bar{j}$	$\bar{q}$	$\bar{t}$	u	u	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	$\bar{q}$	$\bar{t}$	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u		
u	k	h	$\bar{j}$	k	$\bar{t}$	u	$\bar{j}$	h	$\bar{q}$	u	$\bar{j}$	$\bar{q}$	p	q	q	u	$\bar{q}$	u	$\bar{q}$	$\bar{t}$	$\bar{t}$	u	$\bar{q}$	$\bar{q}$	p	q	u	$\bar{q}$	u	$\bar{q}$	$\bar{t}$	$\bar{q}$	u	q		

AND Table.

$*$	a	b	$\bar{b}$	c	$\bar{c}$	d	$\bar{d}$	e	$\bar{e}$	f	$\bar{f}$	g	$\bar{g}$	h	$\bar{h}$	i	$\bar{\imath}$	j	$\bar{j}$	k	$\bar{k}$	l	$\bar{l}$	m	$\bar{m}$	o	$\bar{o}$	p	q	$\bar{q}$	r	$\bar{r}$	s	t	$\bar{t}$	u
a	$\bar{b}$	d	$\bar{c}$	g	$\bar{e}$	d	$\bar{h}$	$\imath$	$\bar{c}$	d	$\bar{h}$	$\bar{l}$	$\imath$	$\bar{k}$	$\bar{k}$	i	$\bar{h}$	i	$\bar{h}$	i	$\bar{h}$	g	g	i	$\bar{h}$	$\bar{l}$	i	$\bar{k}$	i	$\bar{h}$	i	$\bar{k}$	i	$\bar{h}$	g	$\bar{h}$	i	$\bar{k}$
b	d	f	d	j	d	f	j	m	d	f	j	j	m	j	j	m	j	m	j	m	j	j	j	m	j	j	m	j	m	j	m	j	j	j	m	j
$\bar{b}$	$\bar{c}$	d	$\bar{e}$	$\bar{l}$	$\bar{c}$	d	$\bar{k}$	$\imath$	$\bar{e}$	d	$\bar{k}$	g	$\imath$	$\bar{h}$	$\bar{h}$	i	$\bar{k}$	i	$\bar{k}$	i	$\bar{k}$	$\bar{l}$	$\bar{l}$	i	$\bar{k}$	g	i	$\bar{h}$	i	$\bar{k}$	i	$\bar{k}$	$\bar{l}$	$\bar{k}$	i	$\bar{h}$
c	g	j	$\bar{l}$	o	g	j	p	q	$\bar{l}$	j	p	s	q	t	t	q	p	q	p	q	p	o	o	q	p	s	q	t	q	p	q	p	o	p	q	t
$\bar{c}$	$\bar{e}$	d	$\bar{c}$	g	$\bar{e}$	d	$\bar{h}$	i	$\bar{c}$	d	$\bar{h}$	$\bar{l}$	i	$\bar{k}$	$\bar{k}$	i	$\bar{h}$	i	$\bar{h}$	i	$\bar{h}$	g	g	i	$\bar{h}$	$\bar{l}$	i	$\bar{k}$	i	$\bar{h}$	i	$\bar{k}$	g	$\bar{h}$	i	$\bar{k}$
d	d	f	d	j	d	f	j	m	d	f	j	j	m	j	j	m	j	m	j	m	j	j	j	m	j	j	m	j	m	j	m	j	j	j	m	j
$\bar{d}$	$\bar{h}$	j	$\bar{k}$	p	$\bar{h}$	j	$\bar{q}$	q	$\bar{k}$	j	$\bar{q}$	t	q	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	t	q	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u
e	i	m	i	q	i	m	q	r	$\imath$	m	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q
$\bar{e}$	$\bar{c}$	d	$\bar{e}$	$\bar{l}$	$\bar{c}$	d	$\bar{k}$	$\imath$	$\bar{e}$	d	$\bar{k}$	g	$\imath$	$\bar{h}$	$\bar{h}$	i	$\bar{k}$	i	$\bar{k}$	i	$\bar{k}$	$\bar{l}$	$\bar{l}$	i	$\bar{k}$	g	i	$\bar{h}$	i	$\bar{k}$	i	$\bar{k}$	$\bar{l}$	$\bar{k}$	i	$\bar{h}$
f	d	f	d	j	d	f	j	m	d	f	m	j	m	j	j	m	j	m	j	m	j	j	j	m	m	j	m	j	m	j	m	j	j	j	m	j
$\bar{f}$	$\bar{h}$	j	$\bar{k}$	p	$\bar{h}$	j	$\bar{q}$	q	$\bar{k}$	m	$\bar{r}$	t	q	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	r	$\bar{r}$	t	q	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u
g	$\bar{l}$	j	g	s	$\bar{l}$	j	t	q	g	j	t	o	q	p	p	q	t	q	t	q	t	s	s	q	t	o	q	p	q	t	q	t	s	t	q	p
$\bar{g}$	i	m	i	q	i	m	q	r	$\imath$	m	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q
h	$\bar{k}$	j	$\bar{h}$	t	$\bar{k}$	j	u	q	$\bar{h}$	j	u	p	q	$\bar{q}$	$\bar{q}$	q	u	q	u	q	u	t	t	q	u	p	q	$\bar{q}$	q	u	q	u	t	u	q	$\bar{q}$
$\bar{h}$	$\bar{k}$	j	$\bar{h}$	t	$\bar{k}$	j	u	q	$\bar{h}$	j	u	p	q	$\bar{q}$	$\bar{q}$	q	u	q	u	q	u	t	t	q	u	p	q	$\bar{q}$	q	u	q	u	t	u	q	$\bar{q}$
i	i	m	i	q	i	m	q	r	$\imath$	m	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q
$\bar{\imath}$	$\bar{h}$	j	$\bar{k}$	p	$\bar{h}$	j	$\bar{q}$	q	$\bar{k}$	j	$\bar{q}$	t	q	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	t	q	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u
j	i	m	i	q	i	m	q	r	i	m	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q
$\bar{\jmath}$	$\bar{h}$	j	$\bar{k}$	p	$\bar{h}$	j	$\bar{q}$	q	$\bar{k}$	j	$\bar{q}$	t	q	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	t	q	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u
k	i	m	i	q	i	m	q	r	i	m	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q
$\bar{k}$	$\bar{h}$	j	$\bar{k}$	p	$\bar{h}$	j	$\bar{q}$	q	$\bar{k}$	j	$\bar{q}$	t	q	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	t	q	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u
l	g	j	$\bar{l}$	o	g	j	p	q	$\bar{l}$	j	p	s	q	t	t	q	p	q	p	q	p	o	o	q	p	s	q	t	q	p	q	p	o	p	q	t
$\bar{l}$	g	j	$\bar{l}$	o	g	j	p	q	$\bar{l}$	j	p	s	q	t	t	q	p	q	p	q	p	o	o	q	p	s	q	t	q	p	q	p	o	p	q	t
m	i	m	i	q	i	m	q	r	$\imath$	m	r	q	r	q	q	r	q	r	q	r	q	q	q	r	r	q	r	q	r	q	r	q	q	q	r	q
$\bar{m}$	$\bar{h}$	j	$\bar{k}$	p	$\bar{h}$	j	$\bar{q}$	q	$\bar{k}$	m	$\bar{r}$	t	q	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	r	$\bar{r}$	t	q	u	q	$\bar{q}$	r	$\bar{r}$	p	$\bar{q}$	q	u
o	$\bar{l}$	j	g	s	$\bar{l}$	j	t	q	g	j	t	o	q	p	p	q	t	q	t	q	t	s	s	q	t	o	q	p	q	t	q	t	s	t	q	p
$\bar{o}$	i	m	i	q	i	m	q	r	$\imath$	m	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q
p	$\bar{k}$	j	$\bar{h}$	t	$\bar{k}$	j	u	q	$\bar{h}$	j	u	p	q	$\bar{q}$	$\bar{q}$	q	u	q	u	q	u	t	t	q	u	p	q	$\bar{q}$	q	u	q	u	t	u	q	$\bar{q}$
q	i	m	i	q	i	m	q	r	$\imath$	m	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q
$\bar{q}$	$\bar{h}$	j	$\bar{k}$	p	$\bar{h}$	j	$\bar{q}$	q	$\bar{k}$	j	$\bar{q}$	t	q	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	t	q	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u
r	i	m	i	q	i	m	q	r	i	m	r	q	r	q	q	r	q	r	q	r	q	q	q	r	r	q	r	q	r	q	r	q	q	q	r	q
$\bar{r}$	$\bar{h}$	j	$\bar{k}$	p	$\bar{h}$	j	$\bar{q}$	q	$\bar{k}$	m	$\bar{r}$	t	q	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	r	$\bar{r}$	t	q	u	q	$\bar{q}$	r	$\bar{r}$	p	$\bar{q}$	q	u
s	g	j	$\bar{l}$	o	g	j	p	q	$\bar{l}$	j	p	s	q	t	t	q	p	q	p	q	p	o	o	q	p	s	q	t	q	p	q	p	o	p	q	t
t	$\bar{h}$	j	$\bar{k}$	p	$\bar{h}$	j	$\bar{q}$	q	$\bar{k}$	j	$\bar{q}$	t	q	u	u	q	$\bar{q}$	q	$\bar{q}$	q	$\bar{q}$	p	p	q	$\bar{q}$	t	q	u	q	$\bar{q}$	q	$\bar{q}$	p	$\bar{q}$	q	u
$\bar{t}$	i	m	i	q	i	m	q	r	$\imath$	m	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q	q	r	q	r	q	r	q	q	q	r	q
u	$\bar{k}$	j	$\bar{h}$	t	$\bar{k}$	j	u	q	$\bar{h}$	j	u	p	q	$\bar{q}$	$\bar{q}$	q	u	q	u	q	u	t	t	q	u	p	q	$\bar{q}$	q	u	q	u	t	u	q	$\bar{q}$