The Lee Path Connection Algorithm

FRANK RUBIN

Abstract—**The Lee path connection algorithm is probably the most widely used method for finding wire paths on printed circuit boards. It is shown that the original claim of generality for the path cost function is incorrect, and a restriction, called the pathconsistency property, is introduced. The Lee algorithm holds for those path cost functions having this property. Codings for the cells of the grid are proposed which will allow the correct operation of the algorithm under the most general path cost function, using the minimum number of states possible, six states per cell. Then methods for reducing the number of calculations by increasing the number of states are presented.**

Storing computed cell masses is introduced as a method for reducing the amount of calculation for each iteration of the algorithm. Adding the distance from the goal to the path cost function, and expanding the most recently encountered cell, are shown to substantially reduce the number of iterations needed.

Index Terms—**Cell coding, heuristic search, Lee algorithm, path cost function, printed circuit board, rectangular grids, shortest path problem, wire routing.**

I. INTRODUCTION

THE Lee path algorithm [9] is undoubtedly the most widely used algorithm for finding paths in rectangular grids, particularly those involved in printed circuit board wiring. Indeed, most papers in the field describe their routing algorithms as variants or extensions of the Lee algorithm. The Lee algorithm is not inherently restricted to rectangular grids, but is traditionally used for this purpose, and will be so used in this paper because the rectangular grid is the specific case of interest.

The Lee algorithm has the properties that a) it will always find a path if one exists, and b) the path it finds will always have the minimum possible cost. Only algorithms with these properties will be discussed in this paper. Path cost is any measure the user wishes to minimize, and may include length, crossovers with existing wires (if permitted), nearness to other wires, penalties for entering congested areas, or number of vias (in the multilayer case).

Among all known algorithms, Lee's algorithm and its variants require the least storage. There is no matrix of costs, nor are there any successor tables. Its single positional table can be reduced to as little as 2 bits/cell. This is the primary reason for its wide use. For a large printed circuit board may have from 10^4 to 10^5 cells, and sometimes more.

Manuscript received January 22, 1973; revised March 11, 1974.
The author is with the System Development Division, IBM Corporation, Poughkeepsie, N. Y.

In some of the heuristic wiring programs, such as those by Aramaki *et al.* [2] and Fisk *et al.* [5], the Lee algorithm is used as a "last resort" when the heuristic algorithm fails to find a path.

II. SEARCH TECHNIQUES

The Lee algorithm is intended to operate on a very general class of path cost functions called *monotone path functions*. Such a function is a vector $F = (f_1, f_2, \cdots, f_r)$ of individual path cost functions each of which is monotone. That is, if p is a subpath of q, then $f_i(p) \leq f_i(q)$. The entire vector is regarded as having a left-to-right dominance order. That is, $F(p) > F(q)$ if and only if $f_1(p) = f_1(q)$, $f_2(p) = f_2(q)$, $\cdots$, $f_k(p) = f_k(q)$, and $f_{k+1}(p) > f_{k+1}(q)$ for some k, $0 \leq k < r$. This ordering on values of F is also called "lexicographic order."

Let the origin and goal for a particular path problem be given, say o and x. Then the mass of a cell c_n along a path $p = (o, c_1, c_2, \cdots, c_n)$ is defined as $F(p)$, and the mass of the cell c_n is its least mass along all paths.

In its original form, Lee's algorithm uses two lists, L, the list of frontier cells whose masses are known, and L_1, the list of their neighbors. The principal iteration in the algorithm involves the following four steps.

Algorithm 1: Lee Path Algorithm

1) For each admissible (i.e., nonobstacle) neighbor of a cell in L, put the cell and its tentative cell mass in L_1.

2) Adjoin to L all of those cells in L_1 whose mass is minimum. If any of these is the goal, we are done.

3) Delete from L any cell whose neighbors have all been permanently labeled, and clear L_1.

4) If L is empty, no path exists. Otherwise repeat from Step 1.

The preceding procedure uses a rather modest amount of storage. At any given time only the cells on the frontier and their immediate neighbors are stored, along with the information concerning whether a cell has been expanded (that is, the masses of its neighbors have been calculated and placed in list L_1) and what its minimum-cost predecessor had been. The number of cell cost calculations may be reduced by Dijkstra's technique [4] of retaining the cell costs in a separate table and recalculating only the cost of the neighbors of minimum-cost cells selected in Step 2. Since this table would replace the list L_1, there is no increase in storage for this method. Moreover, if one cell at a time is chosen for expansion, lists L and L_1 may be merged into a single list, and Step 3 of Lee's algorithm

is eliminated. A detailed survey of such methods for minimum-path problems appears in [13, ch. 1].

III. MINIMAL-STATE CELL CODING

When Lee's algorithm is applied to a circuit board, the number of cells is extremely large. Therefore, it is desirable to minimize the amount of storage needed for each cell. At least the following must be recorded for each cell: whether the cell is available or an obstacle, and from which direction it was entered along a minimum-cost path. This requires a minimum of five states, one to distinguish unexpanded nonobstacle cells, and four to indicate direction of entry. An obstacle cell could be distinguished from an unoccupied cell simply by assigning it a direction. In the expansion procedure obstacles and previously expanded cells get the same treatment, namely, they are not considered as valid neighbors of a cell being expanded. Thus during expansion they need not be distinguished.

There is another operation, normally ignored in most papers, for which a sixth state should be added. This is the operation of erasing the direction traces so that the next wire path may be found. The sixth state is used to distinguish obstacles due to fixed geometric properties and old wires from cells expanded in the present search. Those cells in the present optimal path are set to state 6 during the path tracing procedure. Then all cells at states 1 to 5 are reset to state 1.

When multiple target cells are possible, as in multipoint nets, a seventh state to indicate a target cell is used.

A major factor in the cost of the algorithm is whether a given cell is placed in list L once or many times. Unless an indication exists in the cell that it has been already placed on the list, the cell may be placed on the list as many times as it is reached. To prevent this, an eighth state "reached but not expanded" is used. This yields a 3-bit cell coding. The list L must now contain three data for each frontier cell:

1) the identity of the cell, that is, its coordinates;
2) the cell mass;
3) the identity of its predecessor cell, or the direction from which the cell was entered.

The corresponding search algorithm is as follows.

Algorithm 2: Modified Lee Path Algorithm

1) Place the initial cell(s) in the list.
2) Find the lowest cost cell c on the list. If it is a target cell, go to Step 12.
3) If c was previously expanded, skip to Step 9.
4) Otherwise generate the first neighbor of c.
5) If the neighbor was previously expanded or it is an obstacle cell, skip to Step 7.
6) Record the cost and predecessor direction of the neighbor and add it to the list.
7) If any more neighbors of c exist, repeat from Step 5 for the next neighbor.
8) Record the direction of c in the cell matrix.
9) Delete c from the cell list.
10) If any more cells exist in L, repeat from Step 2.
11) No path exists. Stop.
12) Trace back along the path to its origin. Done.

This last step is discussed in more detail in Section V, and modifications to reduce the size of the search are presented in Section VII.

IV. DUPLICATE CELL LIST ENTRIES

For the most commonly used path cost functions, including the five mentioned in the introduction, the cost which a cell adds to a path is independent of the other cells in the path. If that cell has been reached along some path and an entry has been made in the cell list, then any subsequent path to that cell must have at least as great a cost. This means that it is not necessary to place a new entry in the cell list.

However, in the general case, a subsequent path to a cell may have lower cost. Adding a second entry to the cell list will increase the time required to search that list for minimum-cost cells. This may be prevented by removing or replacing the earlier entry in the cell list. That would require a search of the list. The time for this search may be reduced by hashing the cell list on the basis of cell coordinates. The hashed list uses more space and takes longer to search for minimum-cost cells. Thus, in general, it has been found less time-consuming to leave the duplicate entries in the cell list.

V. PATH TRACING TECHNIQUES

Most of the earliest shortest path solutions were graph methods designed primarily for hand application. In these methods, the technique for tracing out the minimum path was usually to draw reversed arrows indicating the arc by which a given node was entered. This is not suitable for computer use; however, several variants may be adapted. For each node, the arc by which it was entered could be recorded. It is easily seen that listing the set of arcs by which all nodes were entered also uniquely defines the shortest path, since each arc enters only one node.

When the graph is sparse and the connections are specified by a successor list, then the arc entering a node may be represented by its sequential position in the row of successors of the given node. The entering arc is guaranteed to be present whenever the graph is bilateral (undirected).

The most common method of tracing through a general graph is to specify the predecessor of each node along the shortest path so far. This method is totally general with respect to graph configuration and cost function. For a complete or near-complete graph the method is probably the best possible. The only unessential information represented in this method is the set of predecessors for nodes reached but not on the minimum path to any goal node. Since the predecessor table is normally positional with

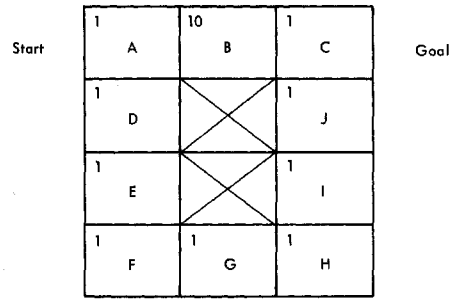

Fig. 1. A path from cell A to cell C is desired. Cell costs are shown in the upper left corner. Moore's path tracing algorithm will not distinguish whether cell B or cell J is the correct predecessor of cell C.

respect to nodes for speed, the storage for these data probably cannot be deleted at reasonable cost, hence supporting the claim of optimality.

Matrix methods for the shortest route problem for all shortest paths between pairs of nodes admit forward tracing by means of a successor matrix. Initially, the successor s_{ik} of node i on the path to node k will be k. When the path from i to k is replaced by a two-part path, i to j and j to k, then s_{ik} will be replaced by s_{ij} (not by j itself). This is possible because at each iteration only one path is being considered for each pair of nodes, not a tree as in the single-origin problem.

Another tracing method requires only that the distance d_i from the origin be recorded for each node. The predecessor of a node j, along the shortest path from the origin, is that node i whose cost satisfies $d_i + c_{ij} = d_j$. This method requires searching all predecessors of each node on the minimum path, and is therefore considerably more costly than other methods. Moreover, the number of bits needed to retain the cost may far exceed the number of bits to represent a node.

Specific types of graphs or cost functions allow specialized tracing techniques. Moore [10] has observed that for the edge cardinality metric, the residue of the number of arcs in the minimum-length path to the given node taken mod 3 is an adequate indicator. This is because the cost of any predecessor of a node of cost c will have cost $c - 1 (\text{mod } 3)$, any successor will have cost $c + 1 (\text{mod } 3)$, and any other neighbor will have cost c. Hence three distinct values serve to differentiate the three classes of neighbors. This method is satisfactory for any graph, but cannot be used with arbitrary arc lengths, for it is not then guaranteed that neighbors which are neither successors nor predecessors will have the same level number.

Fig. 1 illustrates this point. The path desired is from A to C. The cells will have the following level numbers: A-0, B-1, C-2, D-1, E-2, F-0, G-1, H-2, I-0, and J-1. Now B and J have the same level number, so that it is ambiguous whether B or J is the predecessor of C along the minimum-length path.

An even more specialized tracing procedure has been developed by Akers [1] for the edge cardinality metric in rectangular grids. When the grid is rectangular, every neighbor of a cell is either a predecessor or a successor. Thus only two classes of neighbors exist, and only two values are needed to distinguish them. The sequence 0,1,1,0,0,1,1,0,0,$\cdots$ has the needed property that the number following a given position in the sequence is different from that preceding. So this sequence assigned to successive distances from the origin will serve to distinguish predecessors from successors of each cell reached in the expansion process.

The property that every neighbor of a given node is either a successor or predecessor of that node holds whenever all circuits (closed loops) in the graph have even length. Assume the converse. Then there are two neighboring nodes x and y such that $d_x > d_y - 1$ and $d_x < d_y + 1$. Thus $d_x = d_y$. Then the circuit from the origin to x by any shortest path, to y, and to the origin by any shortest path has length $d_x + d_y + 1 = 2d_x + 1$ which is odd, proving the assertion.

The Akers tracing method can be used to get a 2-bit encoding for the cells in the grid, as follows:

00 = available;
01 = obstacle;
10 = reached, with trace bit 0;
11 = reached, with trace bit 1.

Since with edge cardinality metric it is unnecessary to distinguish between cells reached but not expanded and those expanded, the above coding will suffice for representing cells for this special case.

The Akers method, however, cannot be used with arbitrary cost functions. Fig. 2, in which a path from A to F is sought, illustrates this. The level numbers assigned to the cells will be A-0, B-1, C-1, D-1, E-1, F-0. It is now impossible to tell whether C or E is the predecessor of F. Even if C was not expanded and marked with a level number, it would not be possible to tell whether B or D is the predecessor of E.

The inability of the Moore and Akers trace methods to deal with more general cost functions will make them inadequate for many applications. For example, additional cost may be assigned to wires passing through congested

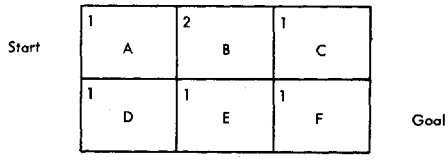

Fig. 2. A path from cell A to cell F is desired. Cell costs are shown in the upper left corner. Akers' tracing algorithm will not show whether cell E or cell C is the correct predecessor of cell F.

areas of the board, close to other wires or fixed obstacles, or which use cells that could otherwise be used for drill-throughs. A discussion of various path cost functions and their uses appears in Rubin [13].

A trace technique presented by Lee [9] is specific to rectangular grids, but can be used with any path cost function whatever. It requires four values. The value associated with a given cell specifies the direction from which it was entered. The four possible directions may be coded in any order. The order—left, right, up, down—has been chosen to enforce the arbitrary decision that routing of a wire shall begin in the horizontal direction.

VI. PATHS WITH MINIMUM TURNS

So far only one demand has been made upon the trace procedure, that it be able to find any least cost path from the goal to the start. There is a second function which could logically be accomplished during this phase of the path procedure. With many cost functions, notably edge cardinality, there is a high likelihood of having many equal-cost paths. In this case one could expect the trace procedure to choose a "best" path according to some additional criterion other than cost. A natural choice at this stage is to select a path which has the fewest line segments, hence the fewest turns.

A means of finding a minimum-cost and minimum-segment path appears to be to add an additional cost factor for turning corners. Unfortunately, the Lee algorithm cannot cope with this type of cost factor. This is because the algorithm (and Lee's proof of it) depends, at least implicitly, upon the path-consistency property.

Definition 1 (Path-consistency property):

Let F be a path cost function, p any minimum-cost path from A to B, and q any minimum-cost path from B to C. If pq is a minimum-cost path from A to C, then F is called *consistent with respect to p and q*. If F has this property for all choices of A, B, C, p, and q, then F is called *consistent.*

Such a property is not obeyed for a path with turn penalties. In Fig. 3, whose path cost function includes cell cost plus a turn penalty of 2, ABE is a minimum-cost path from A to E, EF is a minimum-cost path from E to F, but $ABEF$ is not a minimum-cost path from A to F. $ADEF$ is the unique minimum-cost path from A to F.

Algorithms for dealing with turn penalties have been devised by Caldwell [3], and by Kirby and Potts [8], but they are extremely costly.

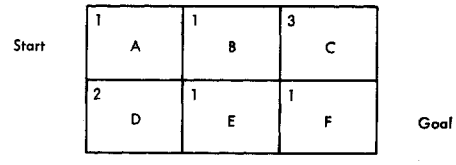

Fig. 3. A path is desired from cell A to cell F using cell cost plus two times the number of turns as the path cost function. Cell cost is shown in the upper left corner. The Lee algorithm gives the incorrect path $ABEF$ because this path cost function does not have the path consistency property.

A means for achieving approximate minimum-segment paths is to control the order in which the neighbors of each cell are generated and considered. Assume that the neighbors of a frontier cell are placed in the cell list in the order in which they are generated, and that the cell list is processed in serial order. The first neighbor generated during the expansion of a cell may be in the same direction as the direction from which that cell was entered. Then the expansion of cells will tend in the same direction as far as possible. This will produce an approximately minimum number of segments in each path. Since the direction is already recorded in the cell list, this method adds very little cost.

VII. REDUCING THE SIZE OF THE SEARCH

One method for reducing the size of the search for a shortest path, the two-ended search technique, has been developed extensively by Pohl [11], [12]. Consider for the moment just the edge cardinality metric, and two cells distant from the edges of the rectangular grid. If the Manhattan distance between the two cells is n, and all cells of distance $\leq n$ from the origin are expanded, then the number of expanded cells will be $2n^2 - 2n + 1$. For large n, this is roughly $2n^2$. However, if a two-ended search is made, about $(n/2)$ levels will be expanded on both sides, so the number of cells expanded will be about $2 \cdot 2(n/2)^2 = n^2$. This is half the number for a one-ended search.

There is a drawback to the two-ended procedure just described. It is necessary in this procedure to distinguish cells expanded from the origin from those expanded from the goal. Then, when a cell which has been previously expanded is encountered, it is possible to determine whether this is a path doubling back upon itself or the completion of the search. Since expanded cells require four values to represent the direction from which they were reached, having two distinguished sets of these val-

ues forces the cell representation to be increased to ten states.

A somewhat lesser benefit can be obtained merely by making a judicious choice of the endpoint from which to begin the search. Whichever endpoint is nearer to one of the four corners of the grid is likely to have a smaller search space because the edges of the grid limit the expansion. When the path cost function is the edge cardinality metric, this reduction is certain.

The major difficulty with both the two-ended and the start-in-the-corner search reduction procedures just mentioned is that the spreading of the frontier takes place in all directions. That is, all cells at a given distance from the origin are expanded whether they are near or far from the goal. In a general graph this may be necessary because there is no sense of whether two nodes are close together or far apart. But in a rectangular grid the Manhattan distance gives an exact measure of how far apart two given cells may be.

Let c_{ij} represent the Manhattan distance from cell i to cell j, and d_{ij} the distance from i to j along the minimum-cost path. Clearly, $d_{ij} \geq c_{ij}$. Assume that the path cost function $F(i) = (f_1(i), f_2(i), \cdots, f_r(i))$ has components of the form $f_k(i) = g_k(i) + a_k d_{oi}$ where $g_k(i)$ is a monotone path function, a_k is a nonnegative constant, o is the origin, and not all a_k are 0. Let x be the goal cell and suppose that the path from o to x passes through the cell i. Then $F(x) = (f_1(x), \cdots, f_r(x))$ where each

$$f_k(x) = g_k(i) + (g_k(x) - g_k(i)) + a_k(d_{oi} + d_{ix})$$

$$= f_k(i) + (g_k(x) - g_k(i)) + a_k d_{ix} \geq f_k(i) + a_k c_{ix}.$$

The term $a_k c_{ix}$ may be regarded as a rough prediction of the cost of the path from i to x in the kth component of the cost function.

Construct a new path function $H(i) = (h_1(i), \cdots, h_r(i))$ to be called the *predicted path cost function* where $h_k(i) = f_k(i) + a_k c_{ix}$. The new function is monotone, for if j is a successor of i then

$$f_k(j) \geq f_k(i)$$

and

$$a_k(d_{oj} + c_{jx}) = a_k((d_{oi} + 1) + c_{jx})$$

$$\geq a_k((d_{oi} + 1) + (c_{ix} - 1)) = a_k(d_{oi} + c_{ix}).$$

There are two questions which must be settled before the use of the predictor function can be reco nmended: Does it give a correct minimum path, and does it reduce the size of the search?

Hart *et al.* [6] have developed a theory of heuristic search which tends to answer the first question affirmatively. Their proof is unduly complex because they are concerned with infinite graphs. In the present work, a broader result is needed because such a wide variety of algorithms is being used. This result is Theorem 1. The second question is answered by Theorem 2 and the ensuing discussion.

Definition 2:

A path algorithm is called *admissible* if, for every consistent monotone path function F, the algorithm will find a minimum-cost path to a goal cell whenever one exists.

Theorem 1:

Let A be an admissible path algorithm and F a monotone path function, as previously shown. Then A applied to the predicted path cost function H constructed from F as above yields a path whose F cost is minimum whenever a path exists.

Proof: Since A is admissible it will find a path whose H cost is minimum whenever one exists. For each k, $1 \leq k \leq r$, $h_k(x) = f_k(x) + a_k c_{xx} = f_k(x)$. So $F(x) = H(x)$ along any path, and thus the path found will also be a minimum-cost path under F.

Definition 3:

The *search space* S_F for a path problem with origin o, goal x, and cost function F is the set of cells i such that $F(i) \leq F(x)$ along minimum-cost paths.

Theorem 2:

The search space for a path algorithm with a predictor is a subset of the search space for the same algorithm without a predictor.

Proof: Let S_F be the search space without the predictor and S_H be the search space with the predictor. Then $i \in S_H$ implies $H(i) \leq H(x)$. But $F(i) \leq H(i) \leq H(x) = F(x)$, so $i \in S_F$ implying $S_H \subset S_F$.

Notice that if a predictive path cost function $H'(i)$ were used, with $h_k'(i) = f_k(i) + b_k c_{ix}$ and $b_k > a_k$, then H' would no longer be monotone, and Theorem 1 would not apply.

Although the search space for the predictor method is smaller, it cannot be immediately established that the actual search is smaller. This is because there may be many synonyms, cells of equal cost, of the goal cell in each algorithm. No order for the expansion of synonyms has been given so far, because the matter has had little significance in the total cost of an algorithm. With the use of the predictor function, the number of synonyms can increase considerably.

For example, consider the edge cardinality metric in the absence of obstacles. Every minimum-length path to a cell in the rectangle determined by the origin and goal, hereafter called the *primary rectangle*, will have the same predicted cost, namely $H(i) = c_{oi} + c_{ix} = c_{ox}$. To find a minimum-cost path it would be wasteful to expand every cell in this rectangle. Indeed, only the cells on such a path need be expanded, for there is no inherent order in the modified Lee algorithm for selecting one equal-cost cell in preference to any other.

Assume that the primary rectangle is nondegenerate,

100	88	75	61	47	33	19	31	45	59	73	86	97	108	119	133	150	169	188	205
89	76	62	48	34	20	9	17	29	43	57	71	84	95	106	117	131	148	167	186
77	63	49	35	21	10	3	7	15	27	41	55	69	82	93	104	115	129	146	165
64	50	36	22	11	4	1	2	6	14	26	40	54	68	81	92	103	114	128	145
78	65	51	37	23	12	5	8	16	28	42	56	70	83	94	105	116	130	147	166
90	79	66	52	38	24	13	18	30	44	58	72	85	96	107	118	132	149	168	187
101	91	80	67	53	39	25	32	46	60	74	87	98	109	120	134	151	170	189	206
112	102										99	110	121	135	152	171	190	207	221
125	113	126	142	161	181	195	176	156	138	123	111	122	136	153	172	191	208	222	235
141	127	143	162	182	200	212	196	177	157	139	124	137	154	173	192	209	223	236	248
160	144	163	183	201	216	226	213	197	178	158	140	155	174	193	210	224	237	249	260
180	164	184	202	217	230	239	227	214	198	179	159	175	194	211	225	238	250	261	272
199	185	203	218	231	243	251	240	228										273	284
215	204	219	232	244	256	262	252	241	253	264	276	287	298	311	323	308	296	285	295
229	220	233	245	257	268	274	263	254	265	277	288	299	312	327	340	324	309	297	307
242	234	246	258	269	280	286	275	266	278	289	300	313	328	344	358	341	325	310	322
255	247										314	329	345	362	375	359	342	326	339
267	259	270	281	291	302	316	332	350	365	347	330	346	363	379	391	376	360	343	357
279	271	282	292	303	317	333	351	368	382	366	348	364	380	395	406	392	377	361	374
290	283	293	304	318	334	352	369	385	398	383	367	381	396	410		407	393	378	390
301	294	305	319	335	353	370	386	401		399	384	397	411				408	394	405
315	306	320	336	354	371	387	402				400							409	
331	321	337	355	372	388	403													
349	338	356	373	389	404														

Fig. 4. Order of expanding cells in a single-ended Lee algorithm search. Comparison of search sizes.

that is, has both dimensions greater than 1. When the origin is expanded, two of its neighbors will be nearer to the goal than it is, and two further away. Choose one of the nearer neighbors. Two of its neighbors will be nearer the goal, and two further away. This process continues until a cell on the boundary of the primary rectangle opposite the origin is reached. Then one neighbor of this cell is nearer the goal and three are further away.

Suppose now that after expanding the origin, one of its neighbors nearer to the goal is chosen. This will define a direction toward the goal. If this cell is expanded next, then its neighbors can be generated in an order so that the next cell to be considered is the neighbor in the direction towards the goal. This procedure can be continued until the opposite boundary of the primary rectangle is reached. If the order for choosing the next cell to expand requires that all neighbors of the last-expanded cell are checked first, then at this point the single neighbor which is nearer to the goal will be chosen, which establishes the preferred direction for the remaining cells.

Since all of the cells which are considered in the preceding procedure are synonyms, the procedure cannot be more costly than the predictor algorithm just presented. The indicated order for expanding the cells will be called *depth-first search*. This term is used in the field of artificial intelligence in a somewhat different sense. There, depth-first search indicates a partial search of a problem space to find any solution path, regardless of cost. Here, only minimum-cost paths will be found, as proven in Theorem 1.

The following algorithm incorporates the aforementioned improvements into the Lee algorithm.

Algorithm 3: Depth-First Predictor Search

1) Place the initial cell(s) on the list. Set the direction entered to 0 and the cost threshold to 0.
2) Find the last cell c whose predicted cost equals the threshold.
3) If none, set the threshold to the least predicted cost of any cell on the list, and repeat Step 2.
4) If c is a target cell, skip to Step 11.
5) If c was previously expanded, skip to Step 8.
6) Otherwise, let d be the direction from which the cell was entered, and consider its neighbors in directions $d + 1, d + 2, d + 3, d + 4 \pmod 4$.
   a) If the neighbor was previously expanded, or it is an obstacle, skip it.
   b) Otherwise record its predicted cost and direction entered at the end of the cell list.
7) Record the direction of c in the cell matrix.
8) Delete c from the cell list.
9) If any more cells exist in the list, repeat from Step 2.
10) No path exists. Stop.
11) Trace back along the path to its origin. Done.

Figs. 4–7 show how the same problem is solved using four variant search procedures. The order in which the cells are expanded is explicitly shown. The size of the search for each of the four methods is summarized:

213	183	151	118	87	58	32	56	85	116	149	181	210	237	262	289	318	350	382	
184	152	119	88	59	33	14	30	54	83	114	147	179	208	235	260	287	316	348	380
153	120	89	60	34	15	4	12	28	52	81	112	145	177	206	233	258	285	314	346
121	90	61	35	16	5	1	3	11	27	51	80	111	144	176	205	232	257	284	313
154	122	91	62	36	17	6	13	29	53	82	113	146	178	207	234	259	286	315	347
185	155	123	92	63	37	18	31	55	84	115	148	180	209	236	261	288	317	349	381
214	186	156	124	93	64	38	57	86	117	150	182	211	238	263	290	319	351	383	
241	215										212	239	264	291	320	352	384		
268	242	269	298	329	362	389	357	324	294	266	240	265	292	321	353	385			
297	270	299	330	363	394		390	358	325	295	267	293	322	354	386				
328	300	331	364	395				391	359	326	296	323	355	387				400	
361	332	365	396					369	392 402	360	327	356	388				401	368	399
393	366	397					370	335										334	367
	398					371	336	303	274	247	222	195	166	194	221	246	273	302	333
					372	337	304	275	248	223	196	167	134	165	193	220	245	272	301
				373	338	305	276	249	224	197	168	135	102	133	164	192	219	244	271
											136	103	72	101	132	163	191	218	243
		374	339	306	277	250	225	198	169	137	104	73	44	70	99	130	161	189	216
	375	340	307	278	251	226	199	170	138	105	74	45	22	42	68	97	128	159	187
376	34	308	279	252	227	200	171	139	106	75	46	23	8	20	40	66	95	126	157
342	309	280	253	228	201	172	140	107	76	47	24	9	2	7	19	39	65	94	125
377	343	310	281	254	229	202	173	141	108	77	48	25	10	21	41	67	96	127	158
	378	344	311	282	255	230	203	174	142	109	78	49	26	43	69	98	129	160	188
		379	345	312	283	256	231	204	175	143	110	79	50	71	100	131	162	190	217

Fig. 5. Order of expanding cells in a two-ended search using Pohl's algorithm. Comparison of search sizes.

						100	104	106	108	110	111	112	116						
					101	73	76	78	80	82	83	84	87	115					
				102	74	48	50	52	54	56	57	58	60	86	114				
			103	75	49	1	2	4	7	11	15	19	23	59	85	113			
			105	77	51	3	5	8	12	16	20	24	27	61	88	117			
			107	79	53	6	9	13	17	21	25	28	30	62	89	118			
			109	81	55	10	14	18	22	26	29	31	33	63	90	119			
											32	34	36	65	92	121			
								120	91	64	35	37	39	67	94	123			
								122	93	66	38	40	42	69	96	125			
								124	95	68	41	43	45	71	98	127			
								126	97	70	44	46	47	72	99	128			
								129											
								130	131	133	136	139	142						
								132	134	137	140	143	145						
								135	138	141	144	146	148						
											147	149	151						
											150	152	154						
											153	155	157						
											156	158	160						
											159	161	162						

Fig. 6. Order of expanding cells in a Lee algorithm search guided by the predictor function. Comparison of search sizes.

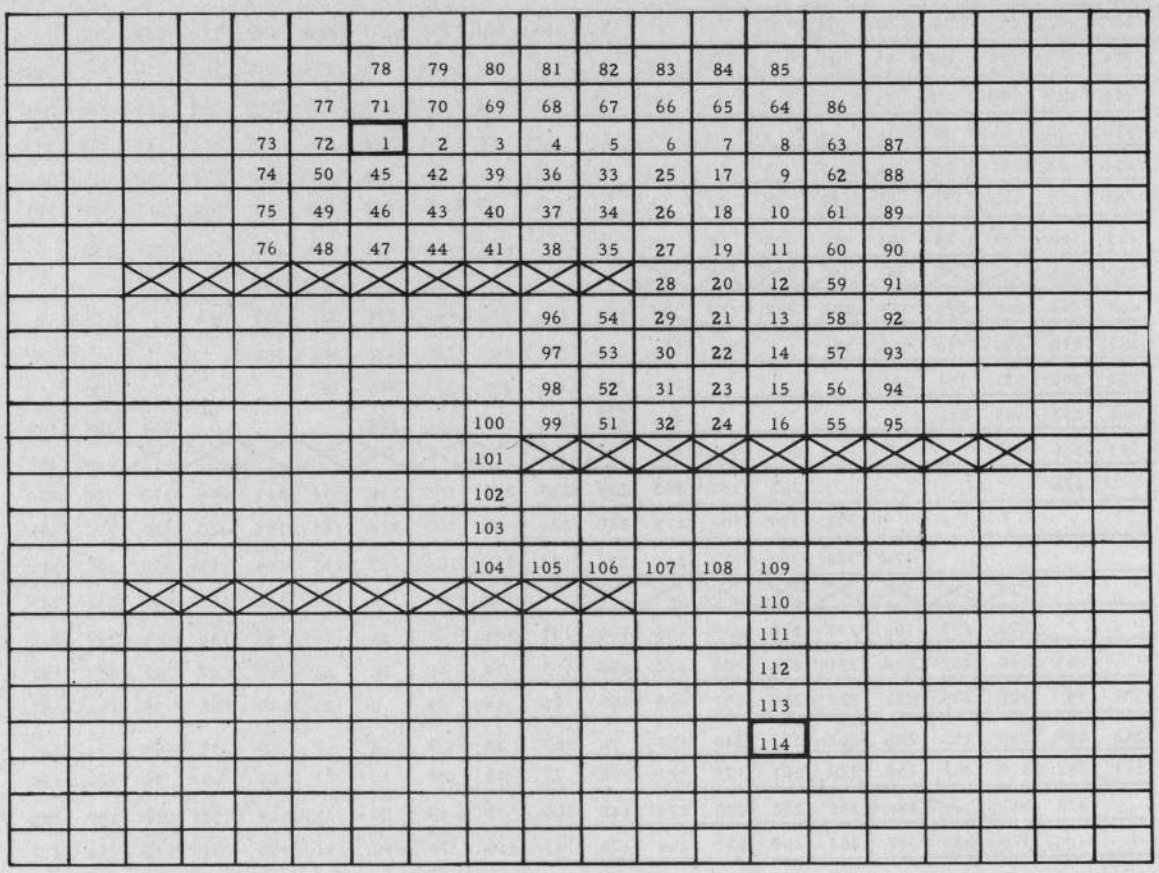

Fig. 7. Order of expanding cells in a depth-first search guided by the predictor function. Comparison of search sizes.

Size of Search	Method
411	Single-ended Lee algorithm.
402	Two-ended Lee search (Pohl).
162	Search with predictor function.
114	Depth-first search guided by the predictor function.

The degree of improvement in the two latter methods is apparent.

REFERENCES

[1] S. B. Akers, Jr., "A modification of Lee's path connection algorithm," *IEEE Trans. Electron. Comput.*, vol. EC-16, pp. 97–98, Feb. 1967.

[2] I. Aramaki, T. Kawabata, and K. Arimoto, "Automation of etching-pattern layout," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 720–730, 1971.

[3] T. Caldwell, "On finding minimum routes in a network with turn penalties," *Commun. Ass. Comput. Mach.*, vol. 4, pp. 107–108, 1961.

[4] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, pp. 269–271, 1959.

[5] C. J. Fisk, D. L. Caskey, and L. E. West, "Topographic simulation as an aid to printed circuit design," in *Proc. SHARE Design Automation Workshop*, vol. 4, pp. 17-1–17-23, 1967.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, pp. 100–107, July 1968.

[7] L. E. Hitchner, "A comparative investigation of the computational efficiency of shortest path algorithms," Univ. Calif., Berkeley, Oper. Res. Ctr. Rep. ORC 68-25, Nov. 1968.

[8] R. F. Kirby and R. B. Potts, "The minimum route problem for networks with turn penalties and prohibitions," *Transp. Res.*, vol. 3, pp. 397–408, 1969.

[9] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comput.*, vol. EC-10, pp. 346–365 Sept. 1961.

[10] E. F. Moore, "Shortest path through a maze," *Proc. Int. Symp. Theory of Switching Circuits*, in *Annals of the Harvard Computation Laboratory*, vol. 30, pt. II. Cambridge, Mass.: Harvard University Press, 1959, pp. 285–292.

[11] I. S. Pohl, "Bidirectional and heuristic search in path problems," Ph.D. dissertation, Stanford Univ., Stanford, Calif., 1969 (Univ. Microfilm 70-1588).

[12] ——, "A theory of bi-directional search in path problems," IBM Corp., Yorktown Hts., N. Y., IBM Res. Rep. RC-2713, Dec. 1, 1969.

[13] F. Rubin, "Printed wire routing for multilayer circuit boards," Ph.D. dissertation, Syracuse Univ., Syracuse, N. Y., 1972 (Univ. Microfilm 73-7767; IEEE-CS Repository 73-211).

Frank Rubin was born in Brooklyn, N. Y., on April 23, 1942. He received the B.S. degree in mathematics from the Massachusetts Institute of Technology, Cambridge, in 1962, the M.S. degree in mathematics from Brandeis University, Waltham, Mass., where he was a National Defense Fellow, in 1964, and the Ph.D. degree in system and information science from Syracuse University, Syracuse, N. Y., in 1972.

He joined IBM Corporation, where he is now a Staff Programmer, in August 1964, working on automatic text handling and automatic flowcharting. This led to automated logic-diagram layout and then to circuit layout and wiring. He has also worked on logic partitioning, on-line update of logic diagrams, computer architecture, processing of natural speech, and several language translators. His other publications are in cryptography, graph theory, and circuit layout and wiring.