2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)

Combining Breadth-First with Depth-First Search Algorithms for VLSI Wire Routing

Xinguo Deng
Center for Discrete Mathematics and Software College,
Fuzhou University
CDM & SW, FZU
Fuzhou, China
xgdeng@fzu.edu.cn

Yangguang Yao,Jia Chen,Yufeng Lin
Software College,
Fuzhou University
SW, FZU
Fuzhou, China
yaoyg@fzu.edu.cn, successjia@126.com,
yufeng_lin@yahoo.com

***Abstract*—A breadth-first search (BFS) algorithm usually needs less time but consumes more computer memory space than a depth-first search (DFS) algorithm to find the shortest path between two nodes. This paper attempts to combine BFS with DFS algorithms to find all shortest paths in the VLSI (Very Large Integration Circuits) wire routing. BFS is used to compute the shortest distance between every position and the start one. DFS is used to traverse all shortest paths in the course of backtracking from the end position to the start one. The effectiveness of the method is proved by the theoretical analysis and the experiment results.**

Keywords-breadth-first serach; depth-first serach; wire routing; shortest paths

I. INTRODUCTION

Wire routing is a computation intensive task in the physical design of integrated circuits. With increasing chip sizes and a proportionate increase in circuit densities, the number of nets on a chip has increased tremendously. Typically, during the physical design of a VLSI (Very Large Integration Circuits) chip, it almost becomes mandatory to run the routing algorithm repeatedly in search of an optimal solution. [1]

Traditionally, routing is done in two stages of global routing and detailed routing sequentially [2, 3]. In the two stage routers, the global router abstracts the details of the routing architecture and performs routing on a coarser architecture. Then, the detailed router refines the routing done by the global router in each channel.

Lee [4] introduced an algorithm for routing a two terminal net on a grid in 1961. Since then, the basic algorithm has been improved for both speed and memory requirements. Lee' s algorithm and its various improved versions form the class of maze routing algorithms.[5,6,7]

As noted in [8], the BFS algorithm of *Wire Routing* can only find one shortest path and the DFS algorithm of *Rat in a Maze* does not guarantee to find a shortest path. A BFS algorithm usually needs less time but consumes more space than a DFS algorithm to find the shortest path between two nodes.[9,10] To sufficiently utilize the respective merit of BFS and DFS algorithms, this paper focuses on combining the BFS and DFS algorithms for VLSI wire routing so as to achieve better wiring quality by simplifying the paths and therefore to reduce manufacturing costs and to increase the reliability. Our objective is to find all of the existing different shortest paths with the same length.

The remainder of the paper is organized as follows. Section 2 introduces the wire routing problem. Section 3 presents in detail the combined BFS and DFS algorithms for VLSI wire routing. Section 4 is the C++ implementation corresponding to the algorithm combining BFS and DFS. Section 5 analyzes the algorithm complexity and provides experimental results compared to previous algorithm. The last section concludes and gives the directions for future work in this field.

II. WIRE ROUTING

A common approach to the wire-routing problem for electrical circuits is to impose a grid over the wire-routing region. The grid divides the routing region into an $n \times m$ array of squares much like a maze.

A maze is a rectangular area with an entrance and an exit. The interior of the maze contains walls or obstacles that one cannot walk through. In our mazes these obstacles are placed along rows and columns that are parallel to the rectangular boundary of the maze. The entrance is at the upper-left corner, and the exit is at the lower-right corner.

A wire runs from the midpoint of one square a to the midpoint of another b. In doing so, the wire may make right-angle turns. Grid squares that already have a wire through them are blocked. To minimize signal delay, we wish to route the wire using a shortest path between a and b.

Suppose that the maze is to be modeled as an $n \times m$ matrix with position (1,1) of the matrix representing the entrance and position (n,m) representing the exit. n and m are, respectively, the number of rows and columns in the maze. Each maze position is described by its row and column intersection. The matrix has a '#' in position (i,j) if there is an obstacle at the corresponding maze position. Otherwise, there is a zero at this matrix position. The rat in a

978-1-4244-6542-2/$26.00© 2010 IEEE

maze problem is to find a path from the entrance to the exit of a maze. A path is a sequence of positions, none of which is blocked, and such that each (other than the first) is the north, south, east, or west neighbor of the preceding position.

III. ALGORITHM COMBINING BFS WITH DFS

To find the shortest path between grid position *a* and *b*, BFS algorithm begins at position *a* and labels its reachable neighbor 1(i.e., they are distances from *a to 1*). Next, the reachable neighbors of the squares with label 1 are labeled 2. This labeling process is continued until the travel either reaches *b* or has no more reachable neighbor. Fig. 1 shows this process for the case *a* =(3,2) and *b* =(5,6). The shaded squares are blocked squares.

Figure 1. An example of BFS

Once the travel has reached *b*, it can be labeled with its distance (8 in the case of Fig. 1). To construct all shortest path(s) between *a* and *b,* the recursive function "DFS" is invoked after the end position is visited.

In the function "DFS", the retracing begins at *b* and moves to any neighbors with the label of one less than *b*'s label. Such a neighbor must exist as each grid's label is one more than that of at least one of its neighbors. In the case of Fig. 1, retracing moves from *b* to (6,6). From here retracing moves to one of its neighbors whose label is one less, and so on until retracing reaches *a*. There are three shortest paths from *a* to *b* in the example of Fig. 1. The programs corresponding to the algorithms of BFS and DFS are detailed in the next section.

IV. C++ IMPLEMENTATION

A. Design

The methodology of top-down modular is adopted to design the program. There are three basic aspects on the problem: input the grid, find all paths, and output all paths. A fourth module "Welcome" that displays the function of the program is also desirable. While this module is not directly related to the problem at hand, the use of such a module enhances the user-friendliness of the program. A fifth module "CalculateMemory" that calculates the memory is necessary here.

B. Program Plan

The design phase has already pointed out the need for five program modules. A root (or main) module invokes five modules in the following sequence: welcome module, input module, find all paths module, output module and calculate memory module.

A C++ program is designed,following the modular structure in Fig. 2. Each program module is coded as a function. The root module is coded as the function "main"; "Welcome", "InputGrid", "FindAllPaths", "BFS", "DFS", "CheckBound", "ShowAllPaths", "ShowOnePath" and "CalculateMemory" modules are implemented through different functions.

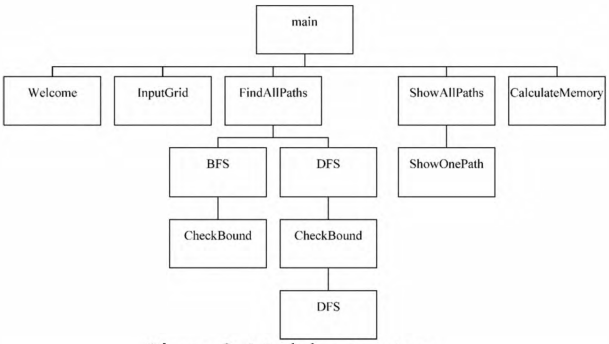

Figure 2. Modular structure

C. Program Development

The function "Welcome" explains the whole C++ program. The function "InputGrid" informs the user that the input is expected as a matrix of "0"s and "#"s except for the start position and the end position. The start position and the end position are inputted with a "S" and an "E" respectively. The size of the matrix is determined first, so the number of rows and the number of columns of a matrix are needed before an input begins. It needs to be determined whether the matrix is to be provided by rows or by columns. In our program, the matrix is inputted by rows, and the input process is implemented by importing the input data from a text file called "in.txt". The function "FindAllPaths" directly calls the functions "BFS" and "DFS", and both of which call the function "CheckBound". The purposes of the rest three functions are obvious and they are self explanatory.

The following Fig. 3 details the function "BFS". The two function parameters sequentially represent the horizontal coordinate, & the vertical coordinate of the start position. At first, the bool flag *found* is initialized false and the queue *q* used to search all shortest paths is emptied. Then the distance of every position to the start position is initialized

-1 which dynamically changes in the course of search.

If the present position is the end, the shortest distance between visited position and the start has already been labeled. The bool flag *found* is changed true. Otherwise its east, south, west and north neighbors are checked sequentially. The next step proceeds if the neighbor not visited is within the bound and is not blocked. The shortest distance from start position to the neighbor of the current position is reserved in the matrix *dis[ ][ ]*. Then the neighbor enters the queue *q*. The next cycle begins if the queue *q* is not empty after all neighbors of the current position are traversed. The minimal distance of the current grid is updated dynamically as the cycle continues. There are two possible results when the cycle terminates because the queue *q* becomes empty. One is that all shortest paths with the same length are found. The other is that there is not a shortest path at all. The two cases are reflected by the bool flag *found*.

```
//BFS computes the shortest distance between every position and the start one.
//(sx,sy): coordinates of the start position
bool Circuit::BFS(int sx,int sy)
{
    bool found=false;
    while(!q.empty()) q.pop(); //clear queue
    memset(dis,0xff,sizeof(dis));//initial value is -1

    cur.x=sx,cur.y=sy;
    dis[cur.x][cur.y]=0;
    q.push(cur);
    while(!q.empty())
    {
        cur=q.front();q.pop();
        //exists shortest path(s)
        if(mat[cur.x][cur.y]=='E')
        {
            found=true;
        }
        //check four directions,enter the queue if traversable
        for(int i=0 ;i<4 ;i++)
        {
            nt=cur;
            nt.x+=dx[i];
            nt.y+=dy[i];
            //within bound & not visited
            if(CheckBound(nt.x,nt.y) && dis[nt.x][nt.y]==-1)
            {   //distance increases one
                dis[nt.x][nt.y]=dis[cur.x][cur.y]+1;
                q.push(nt); //enter the queue
            }
        }
    }
    return found;
}
```

Figure 3. The BFS function

To construct all shortest paths between the start and the end, the recursive function "DFS" illustrated in Fig. 4 is called. In the function, traversal begins from the end to the start. The movement is from the present position to its neighbors whose distance is one less. Coordinates of a converse shortest path are reserved in the temporary array *ans[ ]*. While backing to the start position, the current iterative depth is exactly the shortest distance from a start to the end. Coordinates of a shortest path are reserved in a vector after a cycle. The vector is added to another two dimensional vector representing all shortest paths.

The function "CheckBound" is used to judge whether a position is within the reasonable bound and is not blocked or not. It is invoked by the "BFS" and "DFS" functions. The details of other functions except "BFS" and "DFS" are omitted here. The effects of these functions will be illustrated in the next section.

```
//DFS finds all shortest paths
void Circuit::DFS(int x,int y,int dep)
{
    if(mat[x][y]=='S')
    {
        minStep=dep;

        vector<ipair> onePath;
        onePath.push_back(mp(sx,sy));
        for(int i=dep ;i>=0 ;i--)
            onePath.push_back(ans[i]);
        path.push_back(onePath);    //reserve one path

        return ;
    }
    //check four directions,enter the stack if traversable
    for(int i=0 ;i<4 ;i++)
    {
        int nx=x+dx[i];
        int ny=y+dy[i];
        //retracing
        if(CheckBound(nx,ny) && dis[x][y]-1==dis[nx][ny])
        {
            ans[dep]=mp(nx,ny); //reserve coordinates
            DFS(nx,ny,dep+1); //recursive calling
        }
    }
}
```

Figure 4. The DFS function

V. ALGORITHM COMPLEXITY & EXPERIMENT RESULTS

A. Algorithm Complexity

Since no grid position can get on the queue *q* more than once, it takes O($n \times m$) time (for an $n \times m$ grid) to complete the distance-labeling phase. The time needed for the path-construction phase is O(*PathLen* $\times$ *PathCount*) where *PathLen* is the length of the shortest paths and *PathCount* is the count of different shortest paths with the same length. Thus, the total time complexity of the algorithm combining BFS with DFS is O($n \times m$ + *PathLen* $\times$ *PathCount*).

The worst case occurs when there is no obstacle in the maze at all. The upper bound for the functions "BFS" and "DFS" is theoretically O($n \times m$ + $(n+m-2) \times C(n+m-2,n-1)$) for a maze of $n \times m$ matrix. The more obstacles there are, the less the complexity of the function "BFS" is. The reason is that the count of shortest paths decreases with the increase of obstacles.

B. Experiment Results

Fig. 5 is the output data for the maze of 7×7 matrix in the Fig. 1. The figure shows that the length of shortest paths is 8 and there are 3 different shortest paths from the start to the end in all. Then the coordinates and the map of these paths are output respectively. The next is the final map of these 3 paths overlapped. The last are the total running time and the computer memory occupied.

```
////////////////////////////////////////////////////////////////
    This program combines BFS with DFS algorithms to find all
    shortest paths from the start position to the end one.
////////////////////////////////////////////////////////////////

Enter the row of the gird:
Enter the column of the gird:
Enter the circuit wiring gird
S :the start position
E :the end position
0 :the place we can traverse
# :the obstacle
The length of shortest path(s) is 8.
There are 3 path(s) from 'S' to 'E'
path 1 (coordinates):
2 1
3 1
4 1
4 2
4 3
5 3
5 4
5 5
path 1 (map):
0   0   #   0   0   0   0
0   0   #   #   0   0   0
0   S   0   0   #   0   0
0   1   0   #   #   0   0
#   2   3   4   #   E   0
#   #   #   5   6   7   0
#   #   #   0   0   0   0
path 2 (coordinates):
2 1
3 1
3 2
4 2
4 3
5 3
5 4
5 5
path 2 (map):
0   0   #   0   0   0   0
0   0   #   #   0   0   0
0   S   0   0   #   0   0
0   1   2   #   #   0   0
#   0   3   4   #   E   0
#   #   #   5   6   7   0
#   #   #   0   0   0   0
path 3 (coordinates):
2 1
2 2
3 2
4 2
4 3
5 3
5 4
5 5
path 3 (map):
0   0   #   0   0   0   0
0   0   #   #   0   0   0
0   S   1   0   #   0   0
0   0   2   #   #   0   0
#   0   3   4   #   E   0
#   #   #   5   6   7   0
#   #   #   0   0   0   0
The final map :
0   0   #   0   0   0   0
0   0   #   #   0   0   0
0   S   1   0   #   0   0
0   1   2   #   #   0   0
#   2   3   4   #   E   0
#   #   #   5   6   7   0
#   #   #   0   0   0   0

The total running time is 1 ms
The total memory is 326 KB
```

Figure 5. output data for a the maze of 7×7 matrix in figure 1

The amount of the shortest paths with the same length, the running time and memory consumption of the program vary greatly with the size of the maze and the distribution of obstacles. The worst case occurs when there is not any obstacle in the maze at all. Table I displays part of the statistic data of the worst case tested on the PC of 1G RAM with a 2.20GHz CPU. The table shows that paths, time and memory raise sharply along with the increase of the size of the maze, and the count of the shortest paths is exactly $C(n+m-2,n-1)$. In other words, the shortest distance $n+m-2$ consists of $n-1$ steps downward and $m-1$ steps rightward.

TABLE I. PARTIAL STATISTIC DATA OF WORST CASE

n	paths	C(n+m-2, n-1) (m=n)	time(ms)	memory(KB)
2	2	2	0	326
3	6	6	0	326
4	20	20	15	327
5	70	70	15	330
6	252	252	62	345
7	924	924	218	412
8	3432	3432	703	701
9	12870	12870	2781	1934
10	48620	48620	11625	7163
11	184756	184756	49078	29194
12	705432	705432	207984	121572

Fig. 6 is the partial statistic chart of worst case corresponding to table I. Paths and running time raise evidently along with the increase of n. Fig. 6 indicates that the memory consumption increases slowly when n is less than or equal to a critical value. The reason is that the memory mainly consists of instructions before n reaching this critical value. However, the memory consumption increases sharply when n is greater than the critical value because the memory mainly consists of data thereafter.

Table II is the comparison of partial running time among three different algorithms in the worst case tested on the PC of 1G RAM with a 2.20GHz CPU. The table shows that running the algorithm combining BFS and DFS is obviously faster than running either of the other two algorithms respectively. Please note that data input is neglected and paths are not reserved in table II. The time is just to find the number of all shortest paths.

The reason is that the combined algorithm sufficiently utilizes the respective merit of BFS and DFS algorithms. Namely running a BFS algorithm usually needs less time and running a DFS algorithm often consumes less computer memory space to find the shortest path between two nodes. The essence is that the BFS algorithm avoids the blindness of the DFS algorithm, and using the latter eliminates the redundant paths `produced by using the former in the computer memory.

TABLE II. PARTIAL RUNNING TIME OF THREE DIFFERENT ALGORITHMS IN THE WORST CASE (MS)

n×n matrix	BFS with linear indexing	DFS with pruning & iterative deepening	BFS+DFS	paths
7*7	0	0	0	924
10*10	140	16	16	48620
13*13	6891	1204	1079	2704156
16*16	memory overflow	68750	61422	155117520

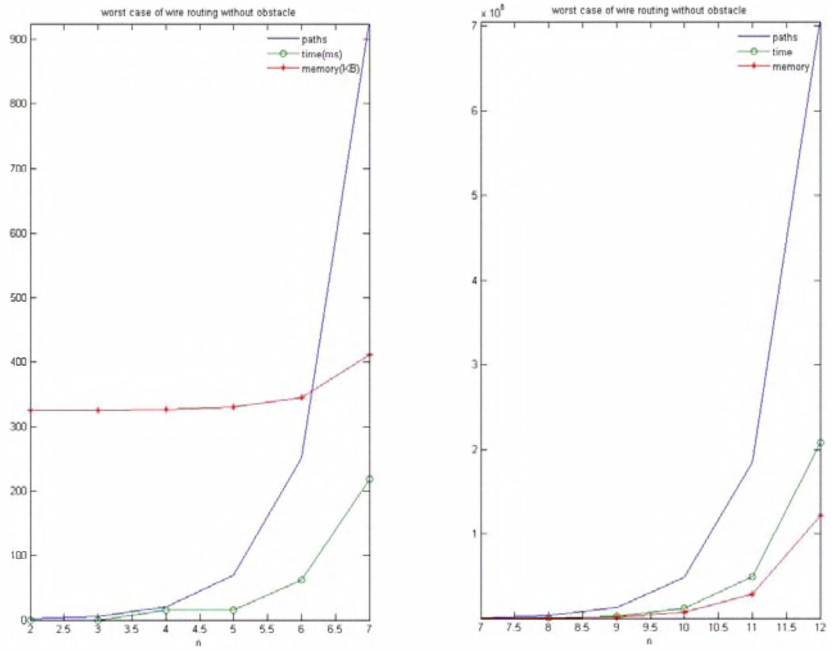

Figure 6. partial statistic chart of worst case corresponding to the table I

VI. CONCLUSION

In order to find all the shortest paths with the same length in the VLSI wire routing, the BFS and DFS algorithms are combined, providing wire routing designers various options to optimize their designs. Therefore, a C++ program is designed and developed to test and implement the ideas of combining these two algorithms. The complexity of the combined algorithm is analyzed theoretically. Further, the satisfactory experiment results of running the C++ program are presented and compared with previous algorithms in this paper. This new method guarantees to find all of the existing shortest paths with the same length in the VLSI wire routing with only moderate computer memory consumption. Hence it has not only theoretical meaningfulness but also great practical usefulness as well as the potential of generating significant economic benefits in the real application field of VLSI wire routing.

Nevertheless, the upper bound of the algorithm complexity is $O(n \times m + (n+m-2) \times C(n+m-2,n-1))$ for a maze of $n \times m$ matrix theoretically. More refinement work of the aforementioned algorithm is needed before its practical application. Future work includes the improvement of the efficiency of the algorithm and further reducing the computer memory consumption.

ACKNOWLEDGMENT

The work was supported by the Natural Science Foundation of Fujian Province under Grant 2009J05142 and the Natural Science Foundation of Fuzhou University under Grant 0220826788.

REFERENCES

[1] Kumar, H.; Kalyan, R.; Bayoumi, M.; Tyagi, A.; Ling, N. Parallel implementation of a cut and paste maze routing algorithm. ISCAS '93, Proceedings. IEEE International Symposium on Circuits and Systems, 1993., Page(s): 2035 - 2038 vol.3

[2] Taghavi, T.; Ghiasi, S.; Sarrafzadeh, M. Routing algorithms: architecture driven rerouting enhancement for FPGAs. ISCAS 2006. Proceedings. IEEE International Symposium on Circuits and Systems, 2006. Page(s): 4 pp. - 5446.

[3] Wayne Wolf, *Modern VLSI Design: System-on-Chip Design,* 3rd ed. ,Pearson Education, Inc., 2003, pp. 518 - 522.

[4] C. Y. Lee, "An algorithm for path connections and its applications," IRE Trans. Electronic Computers, Sep. 1961.

[5] Naveed A. Sherwani. Algorithms for VLSI Physical Design Automation 3rd ed., Kluwer Academic Publishers, 2002, pp. 286 - 288.

[6] Hentschke, R.; Narasimhan, J.; Johann, M.; Reis, R. Maze Routing Steiner Trees With Delay Versus Wire Length Tradeoff. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume: 17, Issue: 8, 2009, Page(s): 1073 - 1086.

[7] Pathak, M.; Sung Kyu Lim. Performance and Thermal-Aware Steiner Routing for 3-D Stacked ICs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume: 28, Issue: 9, 2009, Page(s): 1373 - 1386.

[8] Sartaj Sahni, *Data Structures, Algorithms, and Applications in C++,* 2nd ed.McGraw-Hill,Inc.,2004,pp. 299 - 306.

[9] Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition),* 2nd ed., Publishing House of Electronics Industry. 2009, pp. 401 - 436.

[10] Mitchell Waite, *Data Structures and Algorithms in Java*, 2nd ed., November 16, 2002, pp. 438 - 475.