Improving Depth-First Search Algorithm of VLSI Wire Routing with Pruning and Iterative Deepening*

Xinguo Deng, Yangguang Yao, and Jiarui Chen

College of Mathematics and Computer Science, Fuzhou University, Fuzhou 350108, China
Center for Discrete Mathematics and Theoretical Computer Science, Fuzhou University, Fuzhou 350003, China

Abstract. A depth-first search (DFS) algorithm requires much less memory than breadth-first search (BFS) one. However, the former doesn't guarantee to find the shortest path in the VLSI (Very Large Integration Circuits) wire routing when the latter does. To remedy the shortcoming of DFS, this paper attempts to improve the DFS algorithm for VLSI wire routing by introducing a method of pruning and iterative deepening. This method guarantees to find all of the existing shortest paths with the same length in the VLSI wire routing to provide the wire routing designers more options for optimal designs.

Keywords: Depth-first search, Pruning, Iterative deepening, Wire routing, Shortest paths.

1 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]

As noted in [6], the algorithm of *Rat in a Maze* does not guarantee to find a shortest path from maze entrance to exit. However, the problem of finding a shortest

* The work was supported by the Natural Science Foundation of Fujian Province (No.2009J05142), the Talents Foundation (No.0220826788) and the Scientific & Technological Development Foundation (No.2011-xq-24) of Fuzhou University.

H. Deng et al. (Eds.): AICI 2011, CCIS 237, pp. 100–107, 2011.

path in a maze arises in the VLSI wire routing, too. To minimize signal delay, we wish to route the wire through a shortest path.

The algorithm of *Rat in a Maze* in [6] is essentially a DFS algorithm. To overcome the shortcoming of DFS algorithm, this paper focuses on improving the DFS algorithm 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.

2 DFS Algorithm Improvement

2.1 Weakness of DFS Algorithm

The first path to the end traversed by a BFS algorithm is always the shortest one. A DFS algorithm requires much less memory than BFS one. However, the former doesn't guarantee to find the shortest path in the VLSI wire routing but the latter does.

Assuming figure 1 is a searching tree after optimizing the depth (A is the start position and F is the end position).There are various paths from the start to the end. Among them there is only one shortest path. The following are all possible cases: A→B→D (not a solution), A→B→E→F (not an optimal solution),A→C→E→F (not an optimal solution),A→C→F(the optimal solution). To overcome the blindness of DFS algorithm, the constraints of pruning and iterative deepening are added.

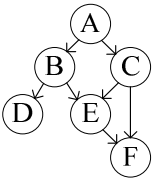

Fig. 1. An example of DFS

2.2 Pruning and Iterative Deepening

Pruning is a technique in machine learning that reduces the size of decision trees by removing sections of the tree that provide little power to classify instances. The dual goals of pruning are reduced complexity of the final classifier as well as better predictive accuracy by the reduction of over fitting and removal of sections of a classifier that may be based on noisy or erroneous data. [7]

Iterative deepening depth-first search (IDDFS) is a state space search strategy in which a depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches d, the depth of the shallowest goal state. On each iteration, IDDFS visits the nodes in the search tree in the same order as DFS, but the cumulative order in which nodes are first visited, assuming no pruning, is effectively breadth-first. [8,9]

2.3 Measures Taken

Pruning can be used to improving the DFS algorithm so that the wire routing can be implemented with great efficiency and speed.

1. Variable *minStep* reprents the shortest distance from a start to the end temporarily. *minStep* is initialized as a very big integer and dynamically changes in the course of search amongvarious existing paths with different lengths between a start and the end. Variable *dep* reprents the distance from a start to the current grid. If *dep*>=*minStep* and the current grid is not the end, the search should backtrack rather than proceed. The reason is that the current path is not one of the shortest paths in this case.

2. *dis[ ][ ]* is a two dimensional array. *dis[x][y]* represents the shortest distance from the start to the present grid temporarily. (*nx*,*ny*) is the neighbor of (x,y). If *dis[x][y]*+1>*dis[nx][ny]*, the search should also backtrack rather than proceed. The reason is the same as in the case 1 above.

3. *matDis [ ][ ]* is a two dimensional array. *matDis[x][y]* represents the Manhattan distance [10] from the current *grid (x,y)* to the end *(ex,ey)*, i.e. |*x* - *ex*| + |*y* - *ey*|. It is the shortest distance between these two grids if there is no obstacle. If *dep*+*matDis[x][y]*>*minStep*, the search should backtrack rather than proceed too. The reason is that the distance of the current path is greater than the previous minimum distance *minStep*.

Pruning can dramatically shorten the search time in an ordinary situation. However, the longest distance from a start to the end is n^2-*1* in a matrix *n*×*n*. The large depth leads to the great search space. Even if the above pruning method is taken, the time complexity is still very large in the worst case.

Iterative deepening works by running depth-first search repeatedly with a growing constraint on how deep to explore the tree. This gives a search that is effectively breadth-first with the low memory requirements of DFS.

Except for pruning, constraint can be added to the distance *dep* between a start and the current grid. The increment of the distance *dep* with an initial value is one in a cyclic search. What's the range of the distance *dep*? It's between the Manhattan distance *matDis[sx][sy]* and the longest distance from a start to the end. The repetition can be broken in advance if all shortest paths with the same length are found.

3 C++ Implementation

3.1 Design

The methodology of top-down modular is adopted to design the program. There are three basic aspects on the problem: input the maze, 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.

3.2 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 by following the modular structure in Figure 2. Each program module is coded as a function. The root module is coded as the function "main"; "Welcome", "InputMaze", "FindAllPaths", "DFS", "CheckBound", "ShowAllPaths", "ShowOnePath" and "CalculateMemory" modules are implemented through different functions.

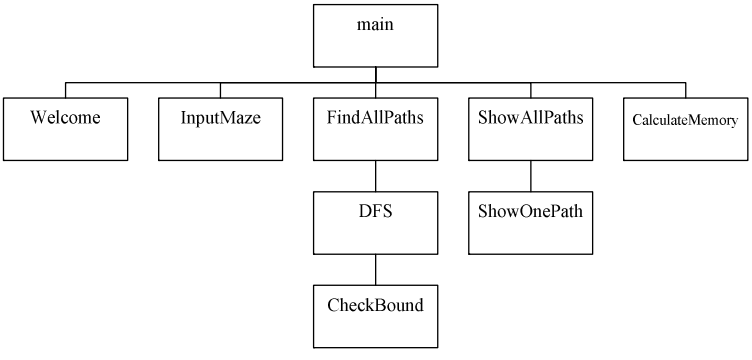

Fig. 2. Modular structure

3.3 Program Development

Function "Welcome" explains the function of the whole C++ program. Function "InputMaze" informs the user that the input is expected as a matrix of "0"s and "#"s besides a start and the end. 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 experiement, 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 idea of IDDFS is embodied in the function "FindAllPaths". At first, the bool flag found is initialized with false. Then a cycle embedded the recursive subroutine DFS begins. The cyclic vabiable i with an increment of one represents the depth limited by IDDFS. It is between the Manhattan distance *matDis[sx][sy]* and the longest distance from a start to the end. The repetition can be terminated ahead when all shortest paths with the same length are found.

The following figure 3 details the recursive function "DFS". The four function parameters represent sequentially the horizontal coordinate, & the vertical coordinate, the distance from the start to the current grid, and the depth limited by IDDFS. The function returns in the following three conditions of pruning. 1. The depth of the current position is greater than the depth controlled; 2. The depth of the current position is greater than the previous minimum length when traversing from the start to the end; 3. The sum of the depth and the Manhattan distance of the current grid is

larger than the depth controlled. Otherwise, the minimum depth of the current grid is updated dynamically.

If the present position is the end, then a path is found. The bool flag *found* is changed to true. The current depth is exactly the shortest distance from a start to the end. To construct a shortest path between the start and the end, traversal begins from the end to the start. The movement is from the present position to its neighbor labeled one less. Such a neighbor must exist as each grid's label is one more than that of at least one of its neighbors. While back to start, coordinates of a shortest path are reserved in a vector. The vector is added to another two dimensional vector representing all shortest paths. The current recursive function returns to the upper one.

If the present position is not the end, its east, south, west or north neighbor is checked sequentially. If the neighbor is within the bound and is not blocked and the depth of the current position after one increment is not larger than the depth of the neighbor, the next step procedes. The visited label of the neighbor is marked. Coordinates of the neighbor are reserved in the matrix of the path. The recursive function "DFS" is called after the depth of the current position is increased by one. This means that the recursive search continues from the neighbor. The label of visited the neighbor is cancelled before the traversal backtracks.

The function "Checkbound" is used to judge whether a position is within the reasonable bound or not. It is called by the recursive function "DFS". The details of the functions "ShowAllPaths", "ShowOnePath" and "CalculateMemory" are omitted here. The effect of these functions will be illustrated in next section.

```
//finds all shortest paths
//coordinates,current depth,depth controlled by IDDFS
void Circuit::DFS(int x,int y,int dep,int curDep)
{
    if(dep>minStep || dep>curDep)   //pruning 1 & 2
        return ;
    if(matDis[x][y]+dep>curDep) //pruning 3
        return ;
    dis[x][y]=dep;  //minimum depth
    if(mat[x][y]=='E')
    {
        found=true;
        minStep=curDep;

        vector<ipair> OnePath;  //vector of one shortest path
        for(int i=dep-1 ;i>=0 ;i--)
            OnePath.push_back(res[i]);
        OnePath.push_back(mp(sx,sy));

        path.push_back(OnePath);    //all shortest paths
        return ;
    }
    for(int i=0 ;i<4 ;i++)  //possible direction
    {
        int nx=x+dx[i];
        int ny=y+dy[i];
        //check bound,not visited,dep+1<=dis[nx][ny]
        if(CheckBound(nx,ny) && !visited[nx][ny] && dep+1<=dis[nx][ny])
        {
            visited[nx][ny]=true;   //visited
            res[dep]=mp(nx,ny); //reserve coordinates
            DFS(nx,ny,dep+1,curDep); //recursive calling
            visited[nx][ny]=false;  //cancell label visited before backtacking
        }
    }
}
```

Fig. 3. The recursive function DFS

4 Algorithm Complexity and Experimental Results

4.1 Algorithm Complexity

For analysis of the algorithm complexity, the worst case is introduced when there is no obstacle in the maze at all. From interior (i.e. non-boundary) positions of the maze, four moves are possible: east, south, west or north. From positions on the boundary of the maze, either two or three moves are possible. However, if the present position is not the end, it is marked "visited" so as to prevent the search from returning here. Thus, the count of moves should be one less. The upper bound for the recursive function "DFS" is $O(a^{2n})$ for a maze of $n \times n$ matrix theoretically ($2<a<3$). The actual upper bound should be less than $O(a^{2n})$ owing to the limit of pruning and iterative deepening. The more obstacles there are, the less the complexity of the recursive function "DFS" is. The reason is that the count of moves decreases with the increase of obstacles.

4.2 Experimental Results

Figure 4 is partial data of the output about a random maze of 17×17 matrix. The figure shows that the length of shortest paths is 38 and there are 160 different paths from the start to the end in all. Then the coordinates and the map of these paths are output respectively. Only one of them is shown below. The data of other 159 paths are omitted here due to the limit of the paper. The next is the final map of these 160 paths overlapped. The last are the total running time and the memory occupied.

The amount of the shortest paths with the same length, running time and memory consumed 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 1 displays part of the statistic data of the worst case. The table shows that paths, time and memory rise sharply along with the increase of the size of the maze.

Table 1. Partial statistic data of worst case

n	paths	time(ms)	Memory (kB)	a^2n (a=2)
2	2	0	326	16
3	6	0	326	64
4	20	15	327	256
5	70	15	330	1024
6	252	62	345	4096
7	924	218	412	16384
8	3432	734	701	65536
9	12870	2859	1934	262144
10	48620	11734	7163	1048576
11	184756	49625	29194	4194304
12	705432	211391	121572	16777216

```
////////////////////////////////////////////////////////////////////
 This program uses the improved Depth-First-Search algorithm to
 find all shortest paths from the start point to the end point.
////////////////////////////////////////////////////////////////////

Enter the row size of gird:
Enter the column of gird:
Enter wiring gird
S :the start point
E :the end point
0 :the place we can traverse
# :the obstacle
The length of shortest path(s) is 38.
There are 160 path(s) from 'S' to 'E'
path 1 (coordinates):
0 0
0 1
0 2
0 3
0 4
0 5
0 6
1 6
1 7
1 8
0 8
0 9
0 10
1 10
1 11
2 11
2 12
3 12
3 13
4 13
5 13
6 13
7 13
7 14
8 14
8 15
9 15
10 15
10 16
11 16
12 16
13 16
13 15
13 14
14 14
15 14
15 15
15 16
16 16
path 1 (map):
S   1   2   3   4   5   6   #   10  11  12  0   0   #   0   0   #
0   0   0   0   0   #   7   8   9   #   13  14  0   #   #   0   0
0   0   #   0   0   #   #   #   #   0   #   15  16  0   0   #   0
0   0   0   #   0   0   0   0   0   0   0   #   17  18  0   0   #
0   0   0   0   #   0   0   #   #   0   0   0   #   19  #   0   0
0   #   #   #   #   #   0   0   #   0   0   #   0   20  0   #   0
0   #   0   0   #   0   0   0   0   #   0   #   0   21  0   0   #
0   #   0   0   0   #   #   0   0   0   #   #   0   22  23  0   0
0   0   0   0   0   0   0   #   0   0   #   #   0   #   24  25  0
0   #   #   #   #   #   #   #   #   #   0   0   0   0   #   26  #
0   0   0   #   0   0   0   0   0   #   0   0   0   0   #   27  28
0   0   0   #   0   0   #   #   #   #   #   0   0   0   0   #   29
0   #   #   #   0   0   0   0   0   0   0   #   0   0   #   0   30
0   0   #   0   0   0   0   0   0   #   #   0   0   0   #   33  31
0   #   0   0   0   0   0   #   #   0   0   0   #   0   34  #   #
0   0   0   0   #   0   0   #   0   0   0   0   #   0   35  36  37
0   0   0   0   #   0   0   #   0   0   0   0   0   0   0   #   E

                                ... ...

The final map :
S   1   2   3   4   5   6   #   10  11  12  13  14  #   0   0   #
0   0   0   0   0   #   7   8   9   #   13  14  15  #   #   0   0
0   0   #   0   0   #   #   #   #   0   #   15  16  17  0   #   0
0   0   0   #   0   0   0   0   0   0   0   #   17  18  0   0   #
0   0   0   0   #   0   0   #   #   0   0   0   #   19  #   0   0
0   #   #   #   #   #   0   0   #   0   0   #   0   20  21  #   0
0   #   0   0   #   0   0   0   0   #   0   #   0   21  22  23  #
0   #   0   0   0   #   #   0   0   0   #   #   0   22  23  24  0
0   0   0   0   0   0   0   #   0   0   #   #   0   #   24  25  0
0   #   #   #   #   #   #   #   #   0   0   0   0   #   26  #   #
0   0   0   #   0   0   0   0   0   #   0   0   0   0   #   27  28
0   0   0   #   0   0   #   #   #   #   #   0   0   0   0   #   29
0   #   #   #   0   0   0   0   0   0   0   #   0   0   #   31  30
0   0   #   0   0   0   0   0   0   #   #   0   0   #   33  32  31
0   #   0   0   0   0   0   #   #   0   0   0   #   0   34  #   #
0   0   0   0   #   0   0   #   0   0   0   0   #   0   35  36  37
0   0   0   0   #   0   0   #   0   0   0   0   0   0   0   #   E

The total running time is 141 ms
The total memory is 373 KB
```

Fig. 4. Partial output data about a random maze of 17×17 matrix

5 Conclusion

In order to find all shortest paths with the same length in the VLSI wire routing, the depth-first search algorithm is improved with the method of pruning and iterative deepening, providing wire routing designers various options to optimize their designs. Therefore, a C++ program is developed to implement the enhancing DFS algorithm. Further, the satisfactory experimental results of running the C++ program are presented. This new method guarantees to find all existing shortest paths with the same length in the VLSI wire routing with only moderate computer memory consumption.

Nevertheless, more refinement work of the aforementioned enhanced DFS algorithm is needed. The upper bound of the algorithm complexity is $O(a^{2n})$ for a maze of $n \times n$ matrix theoretically ($2<a<3$). Even if experimental results show that the actual upper bound is far less than $O(a^{2n})$, the algorithm needs further better before its practical application. Future work includes the improvement of the efficiency of the algorithm and further reducing the computer memory consumption.

References

1. Kumar, H., Kalyan, R., Bayoumi, M., Tyagi, A., Ling, N.: Parallel implementation of a cut and paste maze routing algorithm. In: Proceedings of IEEE International Symposium on Circuits and Systems ISCAS 1993, vol. 3, pp. 2035–2038 (1993)
2. Taghavi, T., Ghiasi, S., Sarrafzadeh, M.: Routing algorithms: architecture driven rerouting enhancement for FPGAs. In: Proceedings of IEEE International Symposium on Circuits and Systems ISCAS 2006, pp. 5443–5446 (2006)
3. Wolf, W.: Modern VLSI Design: System-on-Chip Design, 3rd edn., pp. 518–522. Pearson Education, Inc, London (2003)
4. Lee, C.Y.: An algorithm for path connections and its applications. IRE Trans. Electronic Computers (September 1961)
5. Sherwani, N.A.: Algorithms for VLSI Physical Design Automation, 3rd edn., pp. 286–288. Kluwer Academic Publishers, Dordrecht (2002)
6. Sahni, S.: Data Structures, Algorithms, and Applications in C++, 2nd edn., pp. 268–279. McGraw-Hill, New York (2004)
7. Kantardzic, M.: Data Mining:Concepts, Models, Methods, and Algorithms, 1st edn., pp. 139–164. Wiley-IEEE Press (2002)
8. Ibrahim, A., Fahmi, S.A., Hashmi, S.I., Ho-Jin, C.: Addressing Effective Hidden Web Search Using Iterative Deepening Search and Graph Theory. In: Proceedings of IEEE 8th International Workshops on Computer and Information Technology, CIT 2008, pp. 145–149 (July 2008)
9. Cazenave, T.: Optimizations of data structures, heuristics and algorithms for path-finding on maps. In: Proceedings of IEEE International Symposium on Computational Intelligence and Games, CIG 2006, pp. 27–33 (2006)
10. Dar-Jen, C., Desoky, A.H., Ming, O.Y., Rouchka, E.C.: Compute Pairwise Manhattan Distance and Pearson Correlation Coefficient of Data Points with GPU. In: Proceedings of IEEE International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, SNPD 2009, pp. 501–506 (2009)