© 2011 Yun Wei Chang

SINGLE-LAYER BUS ROUTING FOR HIGH-SPEED BOARDS

BY

YUN WEI CHANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Martin D. F. Wong

ABSTRACT

As the clock frequencies used in industry increase, the timing requirements on high-speed boards become very tight. Since wire length is directly proportional to wire delay of the buses that connect each chip on high-speed boards, each wire in the bus has to be tightly bounded by the maximum and minimum lengths during routing. These rigid requirements cause challenges for automatic routing. Therefore, more aggressive routing algorithms are required for current industrial circuits.

This thesis intends to improve Ozdal and Wong's previous work, which is an algorithmic study of single-layer bus routing on high-speed boards. Their routing algorithm assumes that there are no boundaries in the grid during routing, and the maximum-length bound for each net is always met. This thesis modifies their code so that it does not make those assumptions. As a result, the program can now handle boundaries with wire snaking to meet the minimum-length bound and use diagonal wires if the Manhattan distance between the two terminal pins cannot satisfy the maximum-length bound.

To my parents and friends, for their love and support

ACKNOWLEDGMENTS

I would like to thank my research adviser Professor Martin D. F. Wong, for his ideas, constant guidance and discussions about my thesis project. I would also like to acknowledge Muhammet Mustafa Ozdal for his previous work on single-layer bus routing, which was the foundation of this project. Finally, I would like to thank my parents for funding my study at the University of Illinois. This thesis would not be possible without their support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

As the clock frequencies used in printed circuit boards increase, bus delay has become an increasingly important factor in the performance of high-speed circuits. In a typical bus structure, the data in the bus are clocked into registers. Therefore, all data in different wires of the same bus need to arrive at their destinations approximately at the same time. Otherwise, the clock period needs to be lengthened to ensure all data are ready at their destinations for the registers. Studies show that higher clock frequency requires higher consistency in propagation delay of each wire in the bus [1]. Since the propagation delay is directly proportional to the wire length, a higher degree of wire length matching is needed during bus routing for high-speed boards as well [2]. These tighter wire length constraints raise challenges for automatic routers, and hence require more aggressive routing algorithms.

Ozdal and Wong have been working on developing a large-scale routing system for high-end PCBs [3]. A typical PCB contains several bus structures that connect different modules like memory and input/output, in addition to individual nets. These interconnections need to satisfy rigid timing constraints due to high frequency of the clock on the boards. Ozdal and Wong focus on the single-layer bus routing problem, which assumes that layer assignments for each net have been done and all nets have been routed from their individual pins to chip boundaries [3]. Therefore, the problem is to route all nets between modules belonging to the same bus with the same length constraint. However, there can be interleaving of buses or nets that do not belong to any bus on the PCB boards [3]. For this reason, each net has an individual length constraint.

Ozdal and Wong model the single-layer bus routing problem to be a river routing problem, which has been studied extensively during the past [4]–[6]. In a river routing problem, all terminal pins are aligned on two opposite sides of the circuit. There is an underlying grid where routes go center-to-center

of each cell, and the terminal pins are aligned at the top and bottom rows of the grid. In addition, they added extra wire length constraints on all the nets for routing. Figure 1.1 gives an example of a routing solution of a single-layer bus routing problem that involves extension of wire length produced by Ozdal and Wong's algorithm.

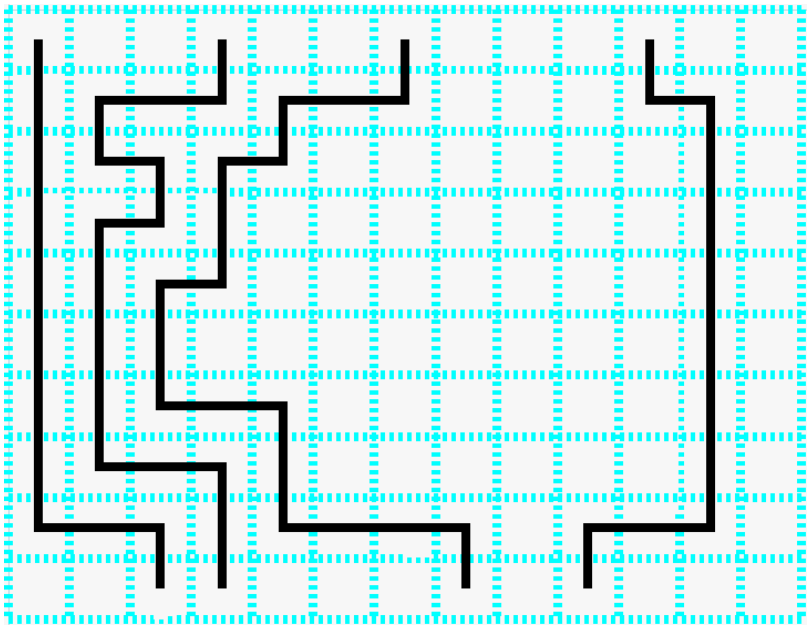

Figure 1.1: Sample routing solution that satisfies the length constraints.

1.1 Minimum-Area Maximum-Length Routing

In order to ensure that the individual length constraint is met, Ozdal and Wong enforce a minimum-area and a maximum-length constraint on each net. The objective is to route each net within maximum-length bound while allocating at least the pre-specified area. Therefore, the nets can be extended in this area during post processing. The shorter nets are allocated with more area for wire snaking meet minimum-length bound. The high-level algorithm for routing with minimum-area maximum-length constraint is shown here [3]:

procedure allocateArea:

find the leftmost boundary L_i for each net i

find the rightmost boundary R_i for each net i

for each net i from left to right
    while Route(L_i, L_{i+1}) violates minimum-area constraint
        flip an appropriate corner of L_{i+1} rightwards

The algorithm first determines the leftmost and rightmost boundaries for all the nets, and the area between the right and left boundaries is where the net must be routed. These boundaries are decided by the previous net and the maximum detour, which is defined by maximum-length bound. Any grid cell that goes beyond the maximum detour is not allocated for the net, so it is not inside the boundary. After all the boundaries have been assigned, the algorithm starts routing the nets from left to right. Each net is routed as close to the previous net as possible while not detouring so much as to violate maximum-length bound. The remaining grid cells are used during post-processing when wire snaking is needed. An example of snaking is given in Figure 1.2. The net is guaranteed to meet the maximum constraint because the left and right boundaries are decided based on the maximum detour, which limits the length of the wire. Thus, the algorithm only checks for minimum-area constraint.

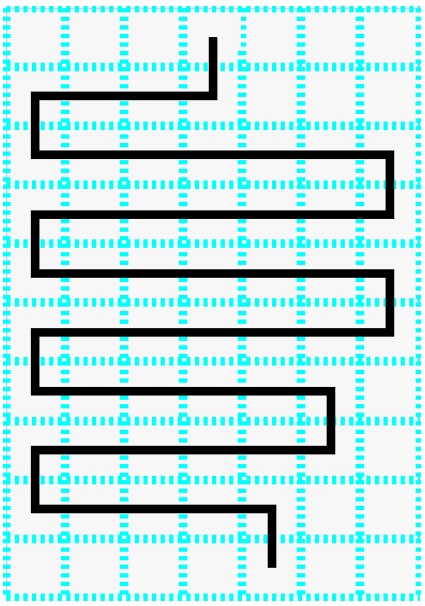

Figure 1.2: Sample example of wire snaking to lengthen the wire.

After one net is first routed, the area is checked against the minimum-area constraint. If the area is below the constraint, the right boundary (L_i) will

need to be flipped towards the right until it satisfies the min-area constraint to leave enough cells for snaking during post processing. However, L_i cannot go beyond the next net's right boundary (L_{i+1}). If L_i cannot be flipped anymore, the net fails the minimum-area constraint. Each time the right boundary is flipped, the program routes the net and checks the area again. The details of boundary flipping are included in Ozdal and Wong's paper [3].

Even though the minimum-area constraint can guarantee that the net will have at least the minimum area available to route, it is difficult to make the transition from minimum-area constraint to minimum-length constraint. Depending on different shapes of boundaries, some cells can never be used to route. An example is displayed in Figure 1.3, which is not from an actual result of the original program. In this example, the minimum-length bound is 40 for both nets. Both nets have the same available area in the figure, but the right net's maximum length is only 30 because the cells marked with X are not routable. The wire will run into a dead end eventually if those grid cells are routed. Therefore, the minimum-length constraint is needed to replace the minimum-area constraint.

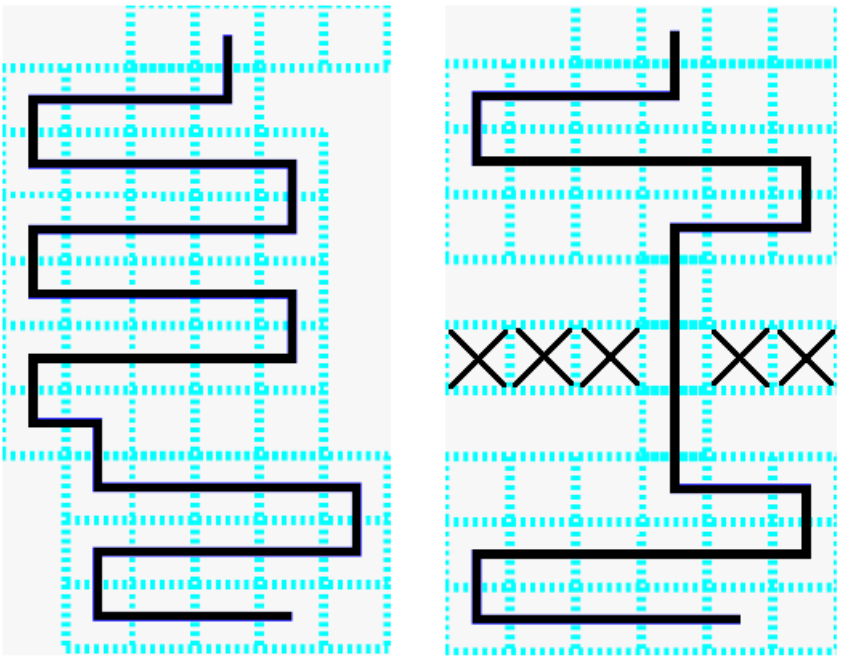

Figure 1.3: A sample case where the same number of available cells can result in different wire lengths.

1.2 Minimum-Length Maximum-Length Routing

The algorithm for routing with minimum-length constraint is almost the same with minimum-area. The difference is that wire length needs to be checked repetitively in all iterations during routing. It was trivial to calculate the total area between right and left boundaries after flipping the right boundary, which is just adding 1 to the available area. However, the calculation for the maximum length within the boundary uses a more complicated method.

To calculate the maximum routable wire length, two vertices, v_k^l and v_k^r, are defined in each row k, with the exceptions of the first and last row. Those two rows only have one vertex, which is the terminal pin. The two vertices in other rows are chosen by comparing the boundaries of the current row with the row above it. The boundary that is closer to the middle of the grid is chosen. For example, if row k has L_i at (3, k) and R_i (7, k) and Row k-1 has L_i (4, k-1) and R_i (8, k-1), the program chooses (4, k) as v_k^l and (7, k) as v_k^r. The two vertices in each row are the turning points for wire snaking.

An edge is defined from each vertex in row k to each vertex in row k+1. Since there are always two vertices for each row, the number of edges from one row to the next is $2 \times 2 = 4$. The length of these edges is defined by the Manhattan distance between the two vertices. For example, if v_k^l = (2, k) and v_k^r = (5, k) in row k, and v_{k+1}^l = (3, k+1) and v_{k+1}^r = (7, k+1) in row k+1, the length of the edge from v_k^l to v_{k+1}^l is $3 - 2 + 1 = 2$. The rest of the edges can be determined in the same way.

For each vertex in the grid, there is a corresponding maximum-distance label for it. This label is defined to be the maximum length from the vertex to the top terminal pin. It is calculated from the topmost row to bottommost row. For each vertex, there are two edges, which connect the current vertex to either the right or left vertex in the upper row. The maximum-distance label of the above vertex is added to the corresponding edge distance to calculate the maximum length. The program compares the maximum length between the left and right vertex in the upper row and sets the larger distance as maximum-distance label of the current vertex. For example, if v_{k-1}^l has maximum-distance label 7 and edge distance 3 to v_k^l, and v_{k-1}^r has maximum-distance label 9 and edge distance 2 to v_k^l, then the maximum distance of v_k^l is set to $max(7 + 3, 9 + 2) = 11$.

The maximum-distance label for the first row is initially set to 0, and

the algorithm continues setting the maximum-distance label for the rest of the vertices until the last vertex is set. The maximum-distance label of the last vertex is the maximum length for the net. Figure 1.4 shows the maximum length corresponding to the left and right boundaries. The red dots correspond to left and right vertex in each row. Note that the first and last rows are not routable in the original program.

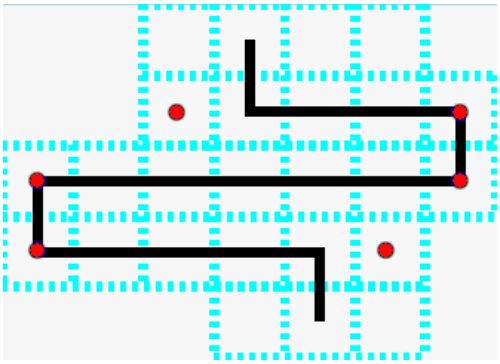

Figure 1.4: A sample case of a net with the maximum length in the allocated cells.

Right before the routing process begins, the router calculates the maximum length achievable with the initially allocated grids by using the process described above. If the maximum length is less than the minimum-length bound, the right boundary needs to be flipped. After the boundary is flipped, the new maximum wire length is checked against the minimum-bound again. If it still does not satisfy the min-bound, the right boundary will be flipped until either the maximum length satisfies the min-bound, or until the right boundary cannot be flipped anymore. If there is enough area for the net to route, the program will call the *find_route* function to do the actual routing for the current net.

1.3 Finding the Actual Route

At this stage, the left and right boundaries, the positions of the vertices in each row and the maximum-distance labels have all been decided. However, the minimum-distance label (D_m), which is also the Manhattan distance to the top terminal pin, is not yet determined for each grid cell inside the

boundary. Maze routing is used to label all the grid cells with the minimum distance because it is suitable for timing-driven global routing [7]-[8]. After all the labeling is done, the router starts from the bottom terminal pin and finds the next cell the net routes to until it reaches the top terminal pin. Typically, there are three choices for the next cell, top, left, and right, assuming that no boundary blocks its way. The program makes the decision by comparing D_m of all three options. The direction with the lowest D_m is chosen. If the distance is the same, the net favors going in horizontal directions over vertical direction. During this process, the program has a variable l for keeping track of the current length of the wire.

After the next cell is chosen, the router adds D_m with the wire length (l) and compares the result with the minimum-length bound. If the result is below the min-length bound, snaking is needed for the current row. It finds the vertex in the upper row that has the longer edge to the current cell and pushes it onto the *route* vector. The program then adds the D_m of the parent vertex and the length of the edge from the current cell to the parent vertex to l. If the result is still less than the min-length bound, more snaking needs to be performed in the remaining rows.

On the other hand, if the result is greater, only partial snaking needs to be used for the current row. Therefore, the parent cell can be any of the cells in the upper row. It is chosen by finding the cell that just satisfies the minimum-length bound. After that, the router can choose the parent cell by following the min path. The chosen parent grid cell is pushed onto the *route* vector, and l is updated to correspond to the parent grid cell. Figure 1.5 gives an example of partial snaking and full snaking. The left net corresponds partial snaking, while the right figure corresponds to full snaking. The algorithm performs this entire process until the net reaches the top terminal pin. The high-level algorithm for the *find_all_route* function is shown below.

procedure find_all_route:
    for each net i from left to right
        find the left and right boundaries that satisfies the
        min-length and max-length bound
        find the left and right vertex on each row and
        calculate their max-length label
        populate the allocated grid cells with maze routing

```
//The actual routing begins here
set current gcell to be the bottom terminal pin
while(current gcell!=top terminal pin)
    choose the parent gcell based on min-distance label
    if(l+min-distance label of parent gcell+1 > min-length bound)
       current gcell=parent gcell;
       update l
       continue;
    //snaking is needed
    if(full snaking is needed)
          choose parent gcell to be the vertex that has longer edge
          update l
    else
          find the parent gcell that just satisfy the min-length bound
          update l
    current gcell=parent gcell;
```

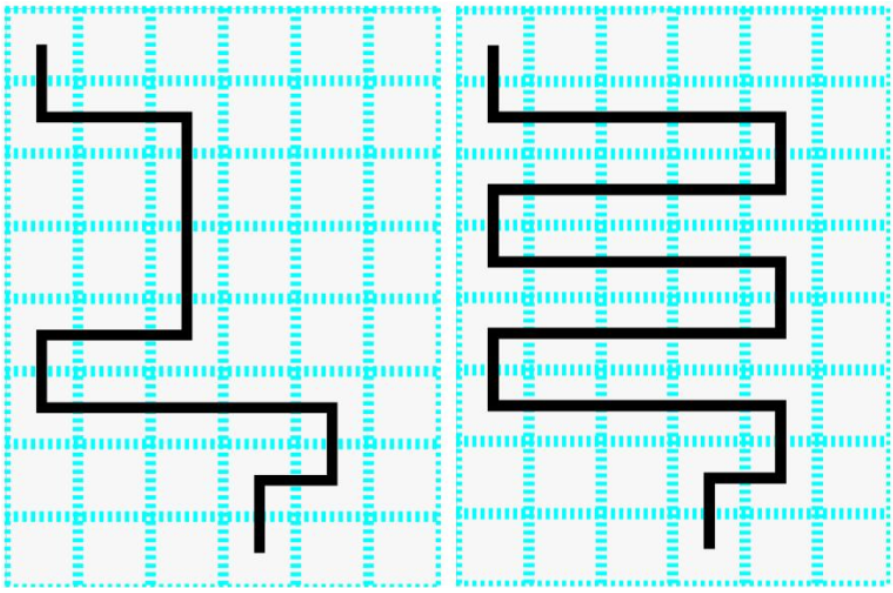

Figure 1.5: A comparison between partial and full snaking.

1.4 Drawing the Actual Route

When the parent grid cells are chosen, they are pushed onto a *route* vector. Each net has its own *route* vector, and they are drawn in the grid after all the nets finish routing. During the drawing process, the background grid cells are already drawn before routing. The grid cells are drawn from the top terminal pin to the bottom terminal pin by reading each element from top to bottom in the *route* vector. If the grid cell is not adjacent to the last cell, it means snaking is needed. The nonadjacent cells are connected by drawing a line going down one cell and turning either left or right, depending on which vertex was chosen during routing.

1.5 Thesis Overview

Chapter 1 provides the background of Ozdal and Wong's work on length-matching algorithm for single-layer bus routing. It discusses the reason for minimum- and maximum-length bounds and the routing process for each net. It gives examples of the program's solution with and without wire snaking and addresses the difference between minimum-area and minimum-length constraints.

Chapter **??** describes how a user can specify a grid's left and right boundaries in boundary files and the two different ways to produce these two files.

Chapter 3 discusses why individual net's right and left boundaries need to be disregarded for minimum-distance labeling. It also introduces a new method to determine the available grid cells for the current net and provides multiple examples of non-routable cells in different directions.

Chapter 4 addresses the problem of unnecessary bending of nets along the left boundary. It introduces the method of giving the parent cell along the last direction priority over other directions if their minimum-distance label is the same. It also discusses how this approach can hurt the routability of the remaining nets such that a cap needs to be used to force the net to turn.

Chapter 5 discusses the effort of making the last row routable by each net and how different mid-points can be chosen to connect nonadjacent cells during routing. It also gives examples of how the maximum length of the wire can be different depending on which mid-point is chosen.

Chapter 6 addresses the need for diagonal wires and how another program, *diagonal*, is used to preprocess the input files and make independent routing in different regions that will be combined into one solution at the end.

Chapter 7 presents two examples with graphs to show the differences between the results of the original program and the new program.

Chapter 8 summarizes the entire thesis and lists a number of improvements that can be made for the single-layer bus routing program.

CHAPTER 2

INTRODUCING LEFT AND RIGHT BOUNDARIES FOR THE GRID

2.1 The Boundary File

The first step to enable Ozdal and Wong's program to handle left and right boundaries is to get the positions of the left and right boundary segments in each row. The boundary information is provided in two boundary files, *lObstacle* and *rObstacle*, which correspond to left and right obstacles respectively. The two boundary files are text files that contain the x coordinates of the corresponding left or right boundary segments of each row. Each x coordinate is separated by a new line, and hence, the number of lines in the boundary file is the same as the height of the grid. The x coordinate of *lObstacle* need to be greater than or equal to 0 and less than the x coordinate of the right boundary in the same row. Likewise, the x coordinate of *rObstacle* need to be greater than or equal to the x coordinate of the left boundary in the same row and less than or equal to the width of the grid. Otherwise, none of the nets will be routable.

2.2 Generating the Boundary File

To generate the boundary files, the user can type in the x coordinates for the left and right boundaries of the grid. However, this method will take too long if there are over hundreds of rows. Therefore, more convenient ways are provided to produce the boundary files. For example, the user can specify a boundary-guidance file, which has segments in the symmetry line of the grid. In this file, the first line contains the distance from the symmetry line to both left and right boundaries. The rest of the lines contain the x coordinates of the segments in the symmetry line. Therefore, the number of lines in the

boundary-guidance file is one more than the actual boundary files. The program produces the left and right boundaries by adding or subtracting the distance from the x coordinate of the segments in the symmetry line in this file. Any value that goes below 0 or above the width of the grid is capped. Figure 2.1 gives an example of a grid produced by a guidance file, which contains distance of 2. The x coordinates of the segments from the first row to the bottom are 3, 4, 4, 4, 5, 5, 4, 4, 3, and 3, which correspond to the block symmetry line. The first two columns are not drawn here because they do not have routable grids.

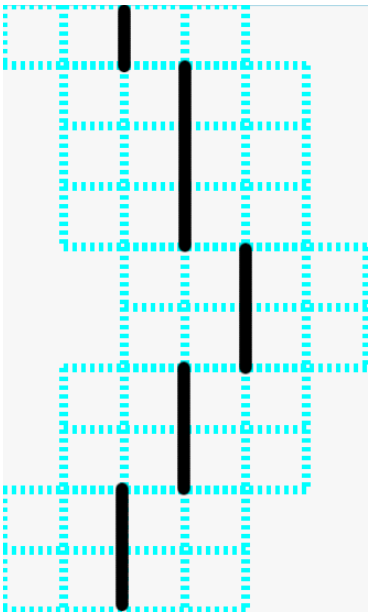

Figure 2.1: A sample grid that is produced by segments of symmetry line.

There is also another way to produce the boundary files. The user can specify contiguous boundary segments together if they have the same x coordinate. The program will read the file and produce the corresponding left and right boundary files. The non-specified regions will be set to default values, which are 0 for the left boundary and the grid width for the right boundary. Since most of the obstacles are rectangular, the boundary will be divided into chunks of different x coordinates most of the time. Therefore, the user can specify the boundaries with fewer entries.

To use these two methods to produce the two boundary files, the user needs to call the diagonal program that the author wrote, which serves as an interface between the user and the *route* program. The usage format

is: *./diagonal input cap boundary filename1 filename2*. The *input* field is the filename of the file that contains all the nets' information. The *cap* field is for limiting zigzag paths, which will be discussed in Chapter 4. If the symmetry line file is used, the filename of the guidance file needs to be given in the *filename* field, and 1 needs to be the argument for the *boundary* field. If the multiple segments files are used, the user needs to specify 2 for the *boundary* field, and include two files, the left-segment and right-segment files, in the filename field. The second filename field is only used if multiple segments files are used, since two files are needed to specify the two boundaries. The diagonal program is also used to determine the dimension and position of the diagonal region. The details will be discussed in Chapter 6.

2.3 Reading the Boundary File

After the boundary files are produced, the author modified Ozdal and Wong's program so it can read the files and generate two boundary vectors that contain the coordinates of the boundaries from top to bottom. The user can tell the program to use the boundary files by adding 1 as an argument after the *route* command in the terminal. If no argument or 0 is given to the program, the program will use the default 0 for the left boundary vector and the grid width for he right boundary vector. Therefore, the user does not need to specify the boundary files if not needed. The two boundary vectors are used during routing to ensure that the nets do not use the grid cells beyond the boundaries. Before the routing begins, the grid cells are drawn, but the cells not inside the boundaries are not drawn. Be aware that the two boundary vectors are for the entire grid and so differ from an individual net's boundary vector.

CHAPTER 3

MINIMUM DISTANCE LABELING WITH LEFT AND RIGHT BOUNDARIES

3.1 Problems with Each Net's Boundary

Now, the program has the grid's right boundary (L^g) and left boundary (R^g), and the grid cells are drawn so any cell not inside the boundary is excluded. However, the program needs to know that the boundaries exist during routing. Therefore, the maze routing that populates the grid cells with minimum-distance label needs to be adjusted to account for the boundaries. Initially, the author tries to cap the left boundary of the first net to be L^g and the right boundary of the last net to be R^g. However, this approach caused many side effects that introduced problems during routing.

One of the problems caused by this approach is an unroutable first or last net. For example, if the first net's right boundary (R_1) has a segment less than L^g, the first net becomes unroutable. The reason is that R_i has to have x coordinate greater than L_i's x coordinate in every row. If one row violates this condition, no grid cell in that row will be routable because no cells are inside the boundary. Since net 1's L_i is modified to be L^g, the net cannot be routable.

In order to solve this problem, R_1 has to be moved to L^g+1 for all segments that are less than or equal to L^g. However, this approach can cause a domino effect where all the net's left and right boundaries need to be adjusted accordingly. In that case, it causes a drastic change to the routing process for the remaining net. The original L_i and R_i, which are chosen based on maximum-length and minimum-length bounds without accounting for L^g and R^g, cannot be used in this case. Therefore, the author decides to make the individual net's right and left boundaries ineffective during routing, except for during the boundary flipping stage of the wire snaking process.

3.2 Available Cells for Each Net

Since L_i and R_i cannot be used during routing, there must be a different way to determine the routable cells in the grid for each net. The new allocation scheme makes all cells inside L^g and R^g available, except for the cells that are blocked by the previous net or the cells that are essential for routing of the remaining nets. This is quite different from Ozdal and Wong's original program, where the nets are confined in the area between L_i and the R_i. Since each net does not depend on L_i and R_i during the routing process, the previous problems are avoided. One additional change caused by this approach is that the program now gives the maximum amount of cells for each net to snake. Therefore, the earlier nets have the most resources and can satisfy the minimum-length constraint more likely. This way, the program can complete routing for as many nets as possible with the given cells until the cells run out.

In the previous version of the program, if the program fails to route, the user only knows which net is not routable. He or she does not have the routing information of previous nets, which are vital to make the adjustment to complete routing for the remaining nets. Therefore, the author decides to change the drawing process so that each net is drawn right after it has been routed instead of drawing all nets after they are all routed. With this new order, even if one net fails the routing process, the previous nets are already drawn in the figure. As a result, the user will be able to see the half-finished routing process, which is the best attempt to finish routing for each net by the automatic routing program. After the user finds out the reason why the program fails to route from the result, he or she can adjust the pin positions or the length constraints in the input file to make all nets routable.

3.3 Maze Routing for Minimum-Distance Label

As mentioned in Section 3.1, the minimum-distance label for each grid cell is populated with the maze routing algorithm. However, the process needs to be updated to accommodate the boundary changes. To determine the new routable grid cells, the program needs to know which cells are used by the previous net, since the left boundary of the current net is basically

the previous net, except for the first left boundary. To store previous net's routing information, a vector is created to store the leftmost and rightmost position in each row for each net during routing. The positions are stored in pairs comprising the x coordinates of the rightmost and leftmost positions from the top to the bottom row. If there is only one cell used in a particular row, the element will contain a pair of the same x coordinate. With this information, the new left boundary for each net (L_i^N) can be determined. Then, the maze routing process has enough information to decide which cells are available for the current net. The pseudo code for the minimum-distance label process is shown below:

```
procedure labelMinDistance:
    create a queue GridCells
    enqueue top terminal pin onto GridCells
    while GridCells is not empty:
        dequeue a cell from GridCells into currentCell
        if the left cell is routable by current net
          relax the left cell
        if the lower cell is routable by current net
          relax the lower cell
        if the right cell is routable by current net
          relax the right cell
```

This piece of code is basically using a breadth-first search for labeling the minimum distance (D_m) for each grid cell like in maze routing. Relax is the function that actually updates D_m and pushes the relaxed cell in the queue. If the grid has been visited before, D_m is updated only if the new distance is less than the old minimum-distance. This is different from the original version of the program, in which D_m is assumed to be the absolute minimum the first time the grid cell is relaxed. Therefore, no cells will be visited again. However, this assumption might not be true if there is more than one route to the target cell, in which case the second route might be shorter than the previous one with different weights of cells. Therefore, the relax function is modified to check D_m of the visited cell and update the label if necessary. This simple change enables the program to handle not only boundaries, but also obstacles in the future. The pseudo code of the

relax function is presented here:

procedure relax(fromCell, toCell):
  if toCell is already visited by the same net
    if toCell's D_m is greater than fromCell's D_m+1
      update toCell's D_m to be fromCell's D_m+1
      enqueue toCell into GridCells
  else
    update toCell's D_m to be fromCell's D_m+1
    enqueue toCell into GridCells

3.4 Decision for Grid Cell Relaxation

How does the maze routing algorithm know which grid cell is available for the current net? Many factors affect this decision: the net index, the position of L^g and R^g in each row, and the cells occupied by previous net. Three directions need to be checked during the labeling process: left, right, and down. For the left direction, the program compares the x coordinate of the left cell (x_c-1) with the previous net's rightmost position in the same row, which is the new left boundary of current net (L_i^N). If x_c-1 is less than or equal to ${L_i}^N$, the program does not relax the left cell. If the current net is the first net, the program checks for the left boundary of the grid, L^g.

3.4.1 Relaxing the lower cell

For the cell below the current cell, the program needs to check: 1. L_i^N and L^g in the next row. 2. The bottom of the grid. 3. Routability of the remaining nets. ${L_i}^N$ and L^g still need to be checked because the lower cell can be outside the boundary. The grid bottom has to be checked because it has to be below the cell below the current cell. The third constraint is to make sure the program leaves enough cells for rest of the nets to route. Therefore, any cell that is essential for them cannot be relaxed. The first two constraints are self-explanatory. The final constraint will be discussed in detail here.

If the current net being routed is the last net, the only constraint is R^g in the next row. For all other nets, the program has to make sure there

are at least n_r cells to the right of the cell below, where n_r represents the number of nets remaining for routing. However, this condition alone does not ensure that the rest of the nets will be routable because R^g can become tighter for the rows below. In that case, more space needs to be allocated for the remaining nets. To account for this situation, a simple decision-making scheme is used. The decision tree is drawn in Figure 3.1.

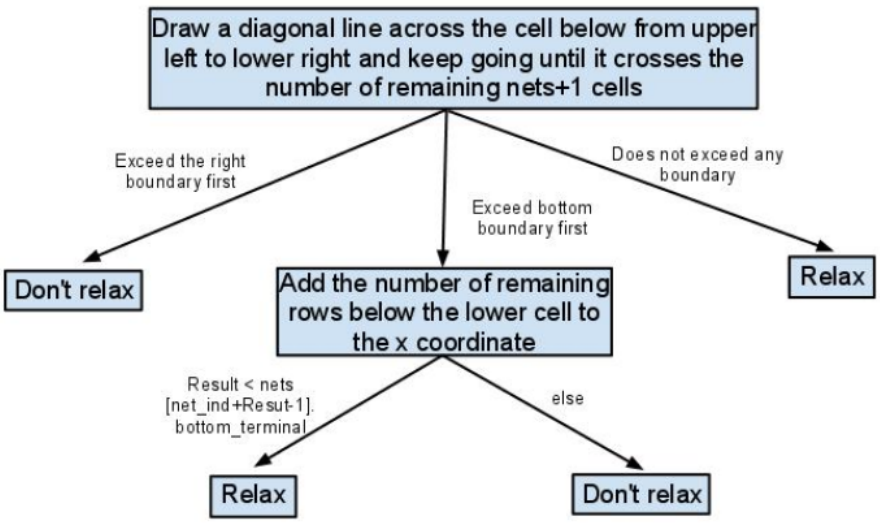

Figure 3.1: The decision tree to decide if the lower cell can be relaxed.

The first decision is to see if the number of rows below the lower cell is less than n_r. If the diagonal line does not exceed the bottom, it means there are enough rows. Then, if the diagonal line exceeds the right boundary before it crosses $n_r + 1$ number of cells, it means the lower cell is essential for the rest of the nets to route. The reason is that there have to be at least n_r cells on the diagonal line to let all the remaining nets route through the right side of the cell. Otherwise, at least one net will be blocked by the previous net during routing on the diagonal line. If the diagonal line does not exceed a boundary, then the lower cell can be relaxed. The rows between $y + 1$ and $y + 1 + n_r$, where y is the y coordinate of the current row, are assumed to have enough cells for the rest of the nets to route. The assumption is true if the y coordinate of the current cell is below $n_r + 1$, since all those cells have been checked by the grid cells above. If the assumption does not hold for the rows above $n_r + 1$, that means it is impossible for all nets to complete routing in those rows. However, the program continues to complete the route for as

many nets as possible until a net fails to route in that region.

An example of a lower cell that cannot be relaxed is shown in Figure 3.2. There are three nets, and the current net is in red. The dots are the terminal pins of the nets. The numbers are the minimum-distance labels for the cells. The process is in the midst of minimum-distance labeling, and the current cell is marked with a red square. The cell below the current cell cannot be relaxed because the diagonal line exceeds the right boundary before it crosses $n_r + 1$ cells, which in this case is 3 cells.

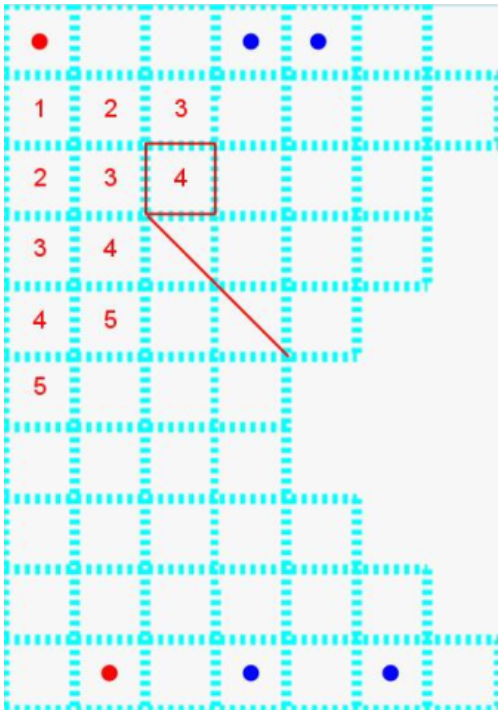

Figure 3.2: An example of how the cell below cannot be relaxed.

For the cells that do not have enough rows below the lower cell for the remaining nets, the router needs to check the bottom terminal pin of one of the remaining nets to see if the lower cell is essential for routing. Depending on the number of rows below the lower cell, different net's bottom terminal pin may be checked. The formula is $i = i_c + r_b + 1$, where i is the net index of the net that the bottom terminal pin needs to be checked, i_c is the index of current net, and r_b is the number of rows below the lower cell. The resulting net's bottom terminal pin (B_i) has to be lower than the x coordinate of the current cell (x_c) plus r_b, to relax the lower cell. If it is not lower, then the cell cannot be relaxed because it is essential for the nets between i and i_c.

Otherwise, not enough cells are allocated for the remaining cell.

An example of an unroutable cell in those rows is shown in Figure 3.3. Again, the current net is marked in red. The dots are the terminal pins, and the numbers are the minimum distance labels. The current cell is marked with a square, and it cannot relax the cell below. The target net, $i = i_c + r_b + 1$, where $i_c = 0$ and $r_b = 1$, has a bottom terminal pin at 4. Therefore, the condition $x_c + r_b < B_i$, where $x_c = 3$, does not hold. As a result, the cell below cannot be relaxed. It can be seen in the graph that the second net will not be routable if the cell is used for net 0 because the path will be blocked by the first net and the bottom terminal pin of the third net.

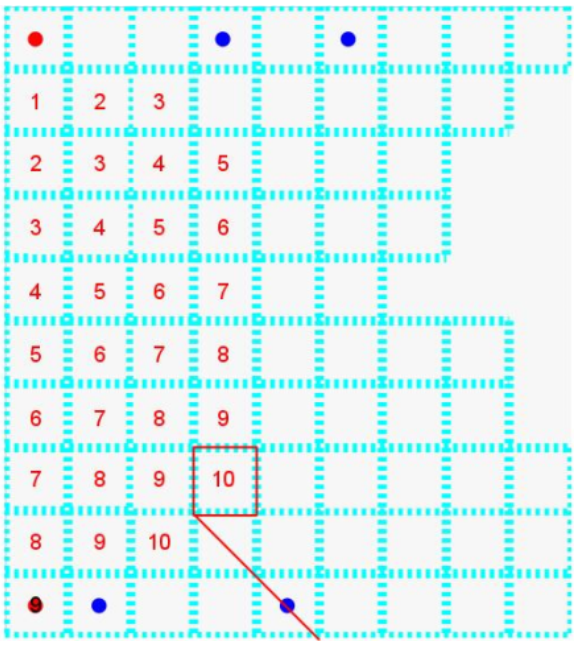

Figure 3.3: An example of an unroutable cell that does not have enough rows below for the remaining nets.

The reason for different relax conditions for cells that have diagonal lines exceeding the bottom boundary is that the terminal pins can block a net's path during routing. Another reason is that the remaining nets do not necessarily cross the diagonal line to complete routing, since their terminal pins can be more to the right than the diagonal line in the bottom row. Therefore, the program checks the terminal pins instead of the diagonal line for the cells that do not have enough rows below.

3.4.2 Relaxing the right cell

The decision whether to relax the right cell is similar to that for the lower cell except that there are two diagonal lines instead of one now. To relax the right cell, the program has to make sure there are at least n_r cells to the right of the right cell. However, this condition alone does not enforce enough cells for the remaining nets to complete the path. The reason is that the grid's right boundary can become tighter in the rows above and below, not leaving enough cells to complete the routing. Therefore, two 45° diagonal lines are drawn extending up and down from the right cell, each crossing n_r cells. All the remaining nets have to route through these two diagonal lines if both lines are in the middle region. If one of these two lines exceeds the right boundary, the right cell cannot be relaxed. However, the rows between the two diagonal lines can have insufficient cells as well. As a result, all rows between row $y_c - n_r$ and row $y_c + n_r$ need to be checked, where y_c is the y coordinate of current cell, to see if there are at least n_r number of cells to the right of the x coordinate of the right cell.

If the upper diagonal line exceeds row 1, an extra relax condition is checked, which checks one of the remaining net's top terminals to see if there are enough cells for the nets in between. The same relax condition for rows 1 to $y_c + n_r$ is still the same. The condition is very similar to that for the lower cell that does not have enough rows below, discussed in Subsection 3.4.1. The formula for the net index is now $i = i_c + y_c$. It does not have the $+1$ like the formula for the bottom boundary because the first row of the grid is not used for routing. The condition is that $x_c + y_c < T_i$, where T_i is the target net's top terminal. If this condition is not satisfied, the right cell cannot be relaxed. The reason is again that the top terminal can block the net's path during routing.

If the lower diagonal line exceeds the bottom boundary, one of the net's bottom terminals is again checked. The same relax condition still holds for rows from $y_c - n_r$ to the bottom row. The formula to calculate the net that the bottom terminal needs to be checked is again $i = i_c + r_b + 1$. The relax condition is $x_c + r_b < B_i - 1$, where B_i is the target net's bottom terminal. The -1 is added because the target cell is $x_c + 1$. The decision tree to relax the right cell is shown in Figure 3.4.

To illustrate, an example of a right cell that cannot be relaxed will be given

for each case here. The current cell will be marked with a square. For the first example, the target net is equal to 1, where $i_c = 0$ and y_c=1. The net's top terminal (T_i) is at 3. In this case, T_i is equal to $x_c + y_c$, where $x_c = 2$ and $y_c = 1$. Since they are equal, the condition does not hold and the right cell cannot be relaxed. The example is shown in Figure 3.5.

For the second example, the right cell cannot be relaxed because the row below the current cell does not have n_r number of cells, which, in this case is 2, to the right of right cell. Notice that all other rows between the two diagonal lines satisfy the condition. The example is shown in Figure 3.6.

For the last example, the right cell cannot be relaxed. The target net is equal to 2, where $i_c = 0$ and r_b=1. The condition $x_c + r_b < B_i - 1$, where $B_i = 4$, $x_c = 2$, and $r_b = 1$, is not satisfied. The example is shown in Figure 3.7.

Figure 3.8 shows all the minimum distance labels for the available grid cells for net 0. Then, the program proceeds to do the actual routing process.

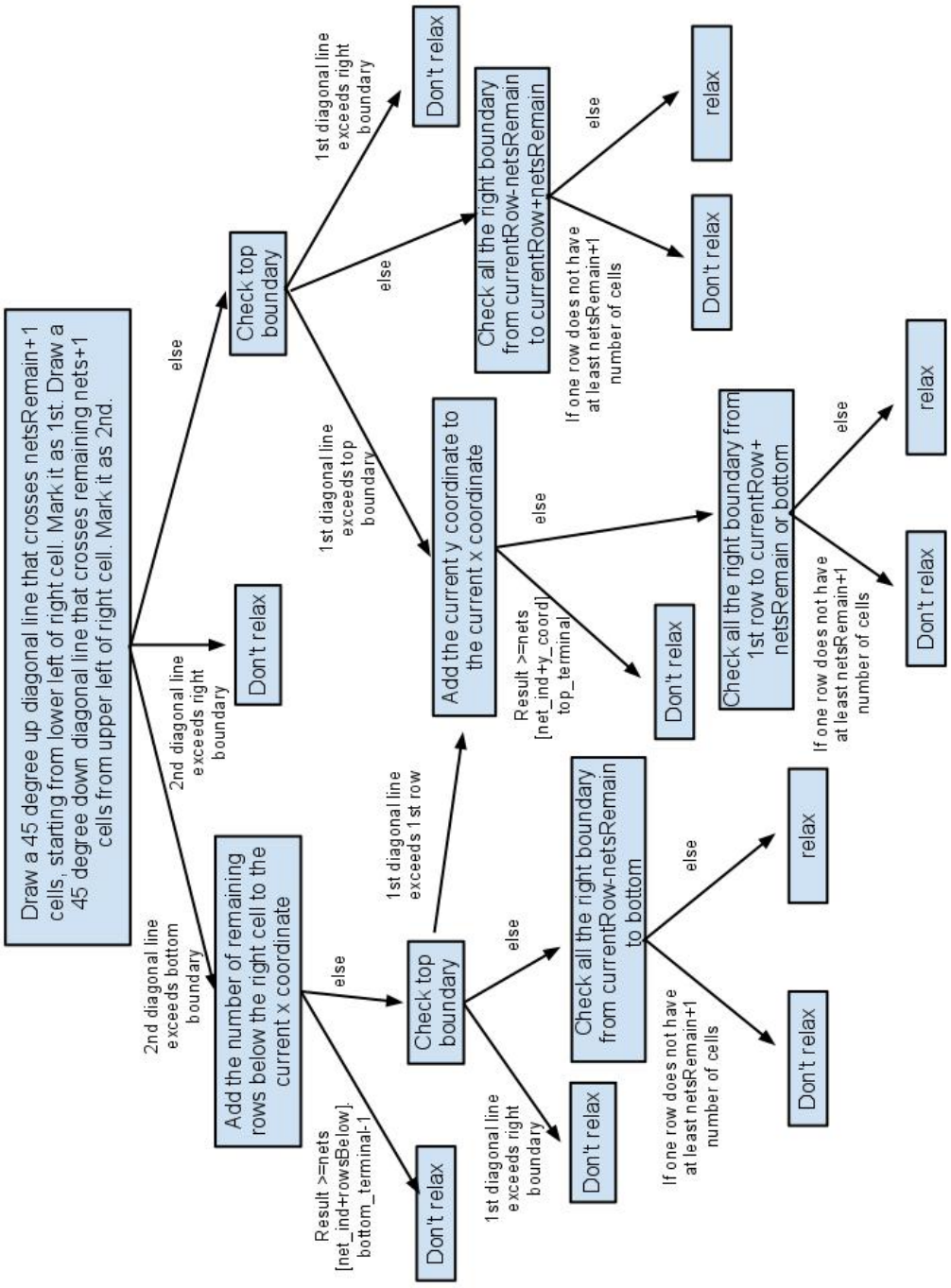

Figure 3.4: The decision tree to decide if the right cell can be relaxed.

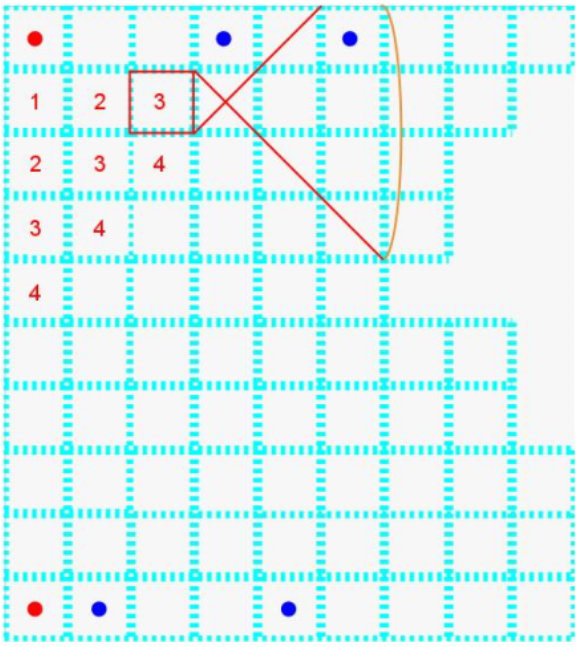

Figure 3.5: An example of an unroutable right cell that does not satisfy the top terminal condition.

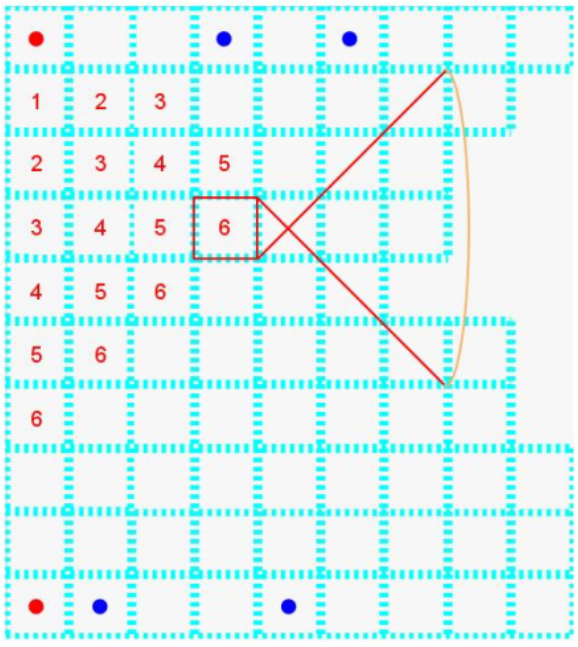

Figure 3.6: An example of an unroutable right cell that does not have enough cells to the right.

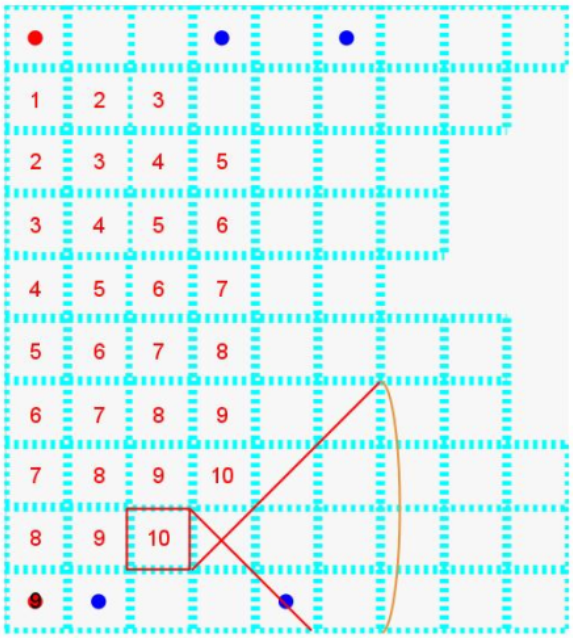

Figure 3.7: An example of an unroutable right cell that does not satisfy the bottom terminal pin condition.

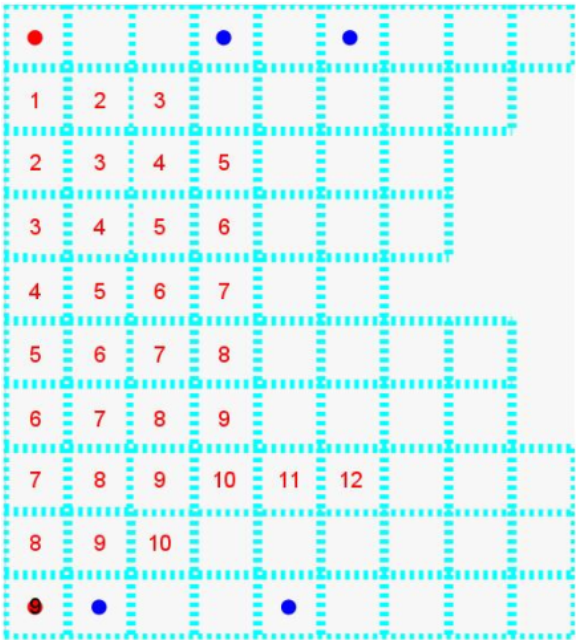

Figure 3.8: Minimum distance label for net 0.

CHAPTER 4

WIRE ROUTING AFTER MINIMUM-DISTANCE LABELING

4.1 Undesirable Zigzag of the Path

With the addition of the grid's left and right boundaries, the grid cells that are beyond those two boundaries cannot be routed. This change causes undesirable zigzag along the left boundary during routing when the left boundary is also in a zigzag shape. An example is given in Figure 4.1. Note that the modified version of the program has blue for the net color to distinguish from the original version.

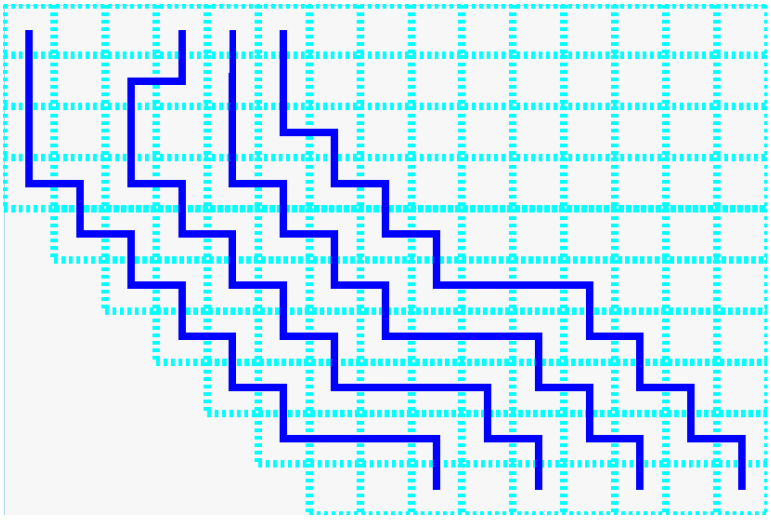

Figure 4.1: An example of excessive bending of a route.

In the original program, the left and right cells have priority over the upper cell. For example, if the upper cell and the left cell have the same minimum-distance label, the left cell is always chosen. Therefore, the net will follow the left boundary as closely as possible. To eliminate unnecessary bending in this case, the priority needs to be changed so the upper cell can

be chosen. However, the priority cannot be changed statically because the upper cell will be chosen over the left cell under every circumstance. If the upper cell is always chosen before the left cell, later nets have more difficulty meeting the minimum-length requirement because there are fewer available cells. Therefore, the priority needs to be changed dynamically.

To eliminate excessive bending, the program gives priority to the previous direction of the net. For example, if up is the previous direction and the minimum distance is not greater than either the left or right cell, the program will choose the upper cell as the next cell. The cell in the previous direction will be chosen until it is blocked by a boundary. As a result, the program can achieve minimum bending for each net. However, there can be a problem with this approach. The nets are routed closer to the left boundary because it leaves the largest area for the following nets to route. To eliminate bending, some of the cells have to be wasted by the current net, and the remaining nets have more difficulty satisfying the minimum-length constraint. If the left boundary zigzags from top left to bottom right, then the up direction will waste cells, which corresponds to Figure 4.1. If the left boundary zigzags from top right to bottom left, then the right direction will waste cells. An example is given in Figure 4.2. Therefore, there must be a cap to limit the number of consecutive segments of the directions that will waste cells to ensure enough area for the remaining nets.

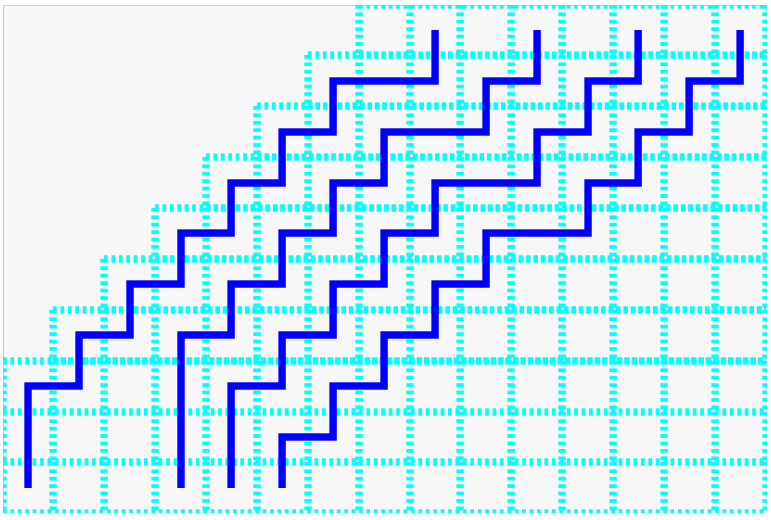

Figure 4.2: An example of excessive bending of a route in another direction.

There is a trade-off between the amount of bending and available cells for

remaining nets. As the cap for the directions that waste cells increases, the bending becomes less because more cells in the same direction are chosen before the net has to turn. However, it also wastes more area for the remaining nets. Figure 4.3 has the same nets as Figure 4.1, but with a cap value of 5. Figure 4.4 has the same nets as Figure 4.2, but with a cap value of 5 also. Since snaking is not needed for the rest of the nets, this cap value is acceptable.

A case where higher cap value hurts routability is shown in Figure 4.5. The third net from the left does not have enough space for snaking to satisfy the minimum length bound. To solve this problem, the user can run the program with a different value in the *cap* field in the command line. In Figure 4.6, the cap value is adjusted to 3, allowing the third net to use more area for snaking during routing. If the user does not specify a value, a heuristic will be used to set the default value for the cap. If the height of the grid is less than 10, the cap will be set to 3. If the height is greater than or equal to 10 but less than 20, the cap will be set to 4. For height greater than 20, the cap is set to 5. Smaller height will require a smaller cap because there is less area. The highest cap value is set to 5 because a value greater than 5 can waste a lot of area. After all, the user can adjust the value manually if the default value produces an undesirable result.

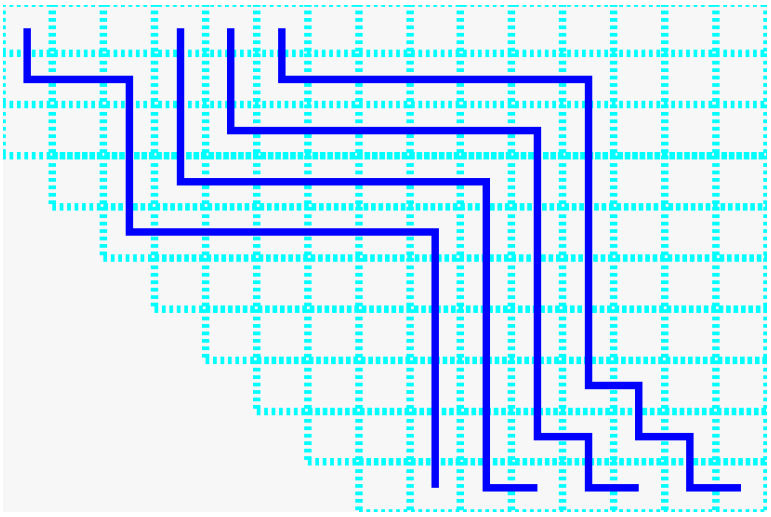

Figure 4.3: An example of the same nets with cap value of 5 from Fig 4.1.

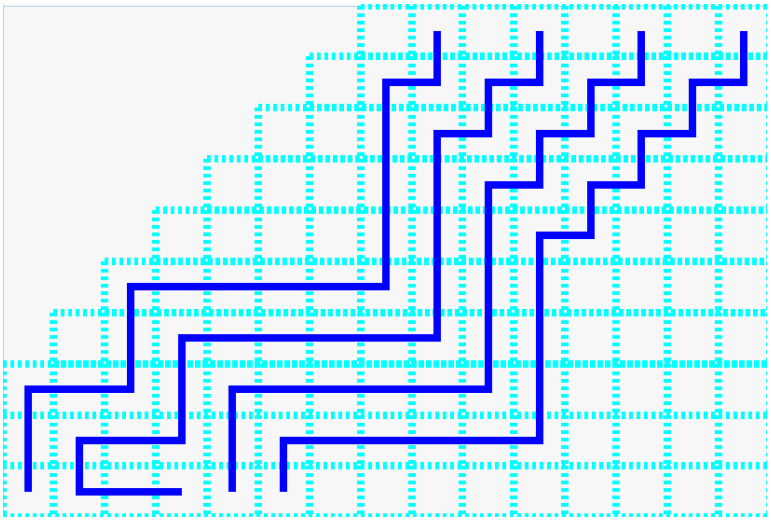

Figure 4.4: An example of the same nets with cap value of 5 from Fig 4.2.

4.2 Routing Nets Closer to Previous Net to Save More Area

In Figure 4.3 and Figure 4.4, some small zigzags appear close to the wire terminal pins. These zigzags are due to the fact that the wires are routed as close to the previous net as possible. The nets are routed close together to save more area for snaking. In addition, the resulting wires look more like a bundle, which looks more like a bus. Therefore, only the first net needs to be adjusted to have less-bending wires. It serves as a guidance-net and the rest of the nets automatically follow it as closely as possible.

If minimum path is always chosen instead of the path that routes closer to the previous net, the remaining nets will have less area for snaking. An example is given in Figure 4.7. The minimum path of net 2 is far apart from net 1, which wastes a lot of grid cells in between. Therefore, a longer path should be taken to let the remaining nets have as many grid cells as possible. As long as the wire length satisfies the maximum-length constraint, the program can take advantage of all extra wire length to route closer to the previous net. An example is given in Figure 4.8. It can be seen that the last net has more grid cells for snaking compared to Figure 4.7.

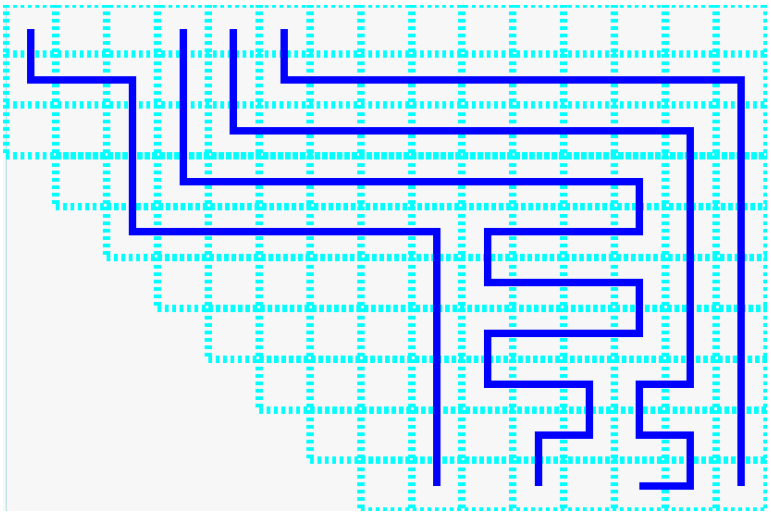

Figure 4.5: An example of how a big value of cap wastes area for snaking.

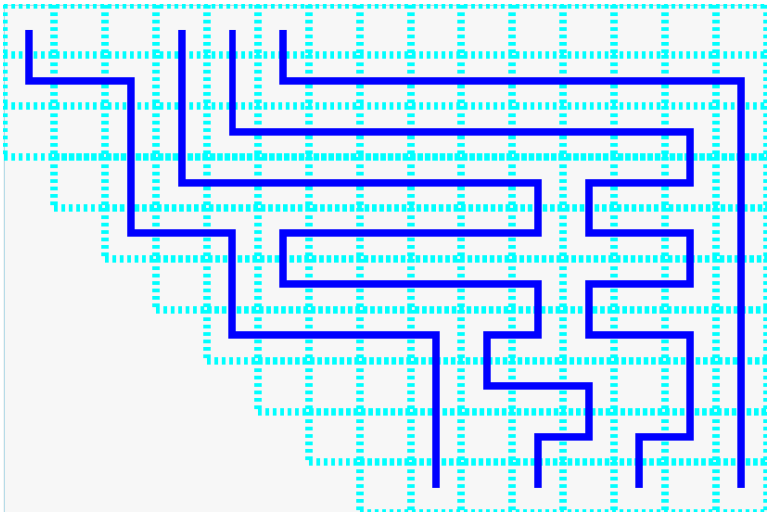

Figure 4.6: A smaller cap allows remaining nets to snake with more area.

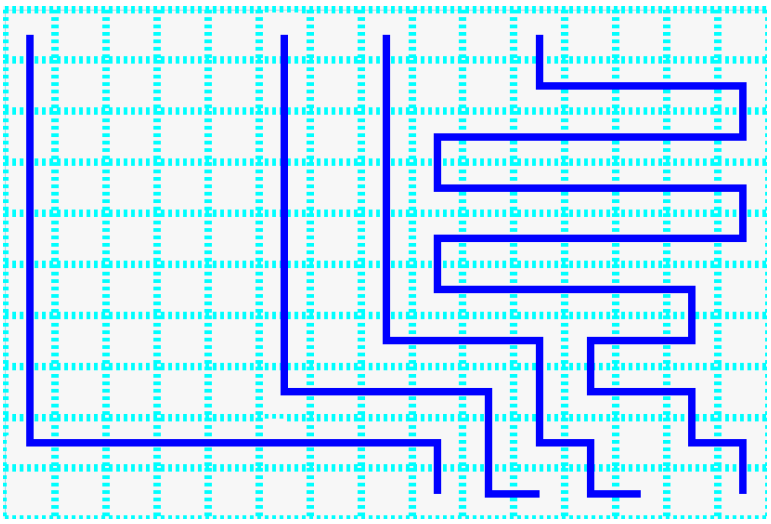

Figure 4.7: An example of how minimum path wastes area for snaking.

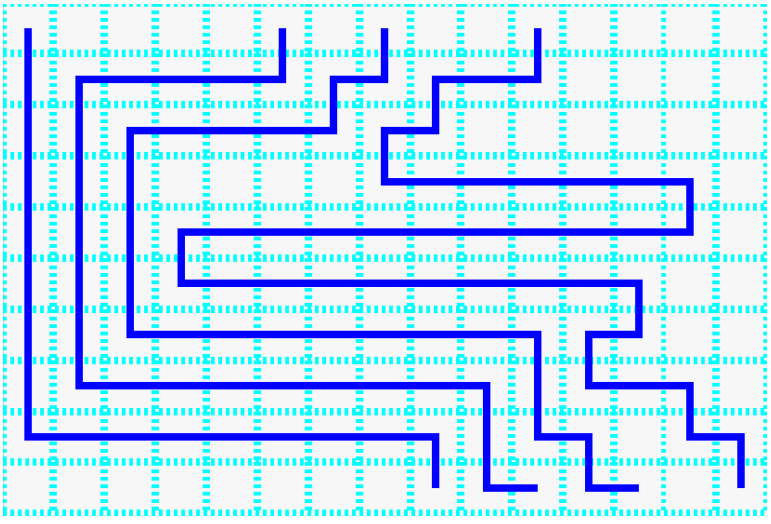

Figure 4.8: A longer route allows the remaining nets to snake with more area.

CHAPTER 5

WIRE SNAKING WITH LEFT AND RIGHT BOUNDARIES

5.1 Adding One More Option for Middle Point for Wire Snaking

With the addition of the grid's left and right boundaries, the vertices in each row, which are assigned based on the old boundary L_i and R_i, need to be adjusted if they are outside the grid's right or left boundary. The first and last rows only have one vertex, which is the terminal point. A lot of grid cells are wasted in that case. Therefore, we change the program so the wires can route horizontally in the last row during routing, which gives more area for snaking. In Figure 5.1, the longest wire without using the last row has length 46, which is shown at the left. The maximum wire length with routable last row is 52, which is shown at the right.

To make the last row routable and achieve more available grid cells, the snaking part has to be modified. During the routing process, each chosen grid cell is pushed onto the *route* vector from the bottom terminal pin to the top terminal pin. If snaking is needed during the process, the vertex with longer edge in the upper row is chosen instead of the next adjacent cell. Since these two cells are not adjacent to each other, the path between them becomes ambiguous.

In the original program, the program always chooses one middle point to connect the two cells. The middle point is chosen to be one unit above the current cell. For example, if the current cell has the coordinates (x_1, y) and the left vertex in the upper row has the coordinates (x_2, y-1), the middle point is always chosen to be (x_1, y-1). After the mid-point is chosen, the program draws one line from the current cell to the mid-point and another line from the mid-point to the vertex in upper row. Therefore, only one turn is needed to connect the two cells. However, there is another mid-point that

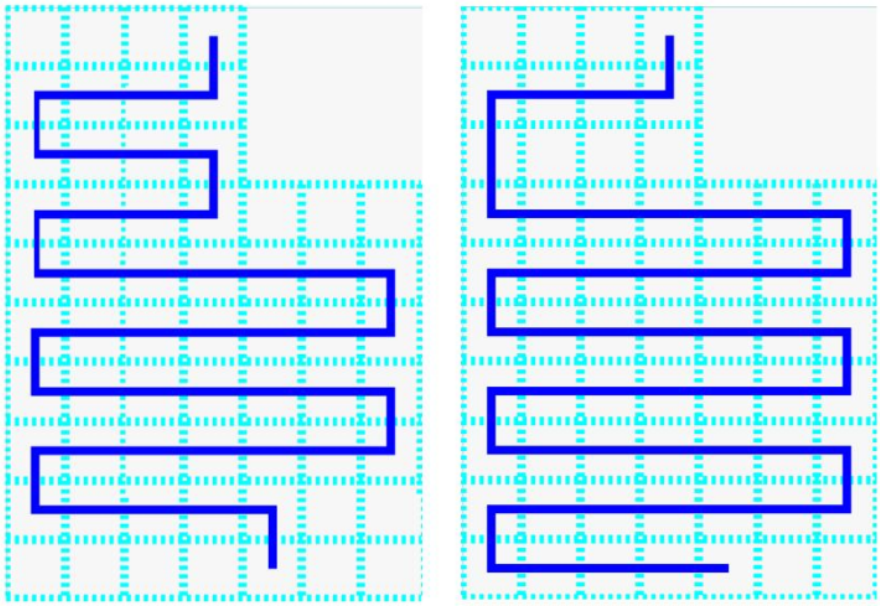

Figure 5.1: The comparison of routable and unroutable last row.

has only one turn, which is (x_2, y). By adding this option for the mid-point, the snaking becomes much more flexible. Otherwise, the last row cannot be used for snaking because the mid-point will never be in the bottom row.

Figure 5.2 labels with red dots the vertices chosen during snaking from the example in Figure 5.1. The resulting wire at the right gains 6 more units due to the choices of the other mid-points in the rows below. These extra wires are circled in black. The right vertex in row 3 does not need to get pushed towards center because it does not run into a dead end. Unlike the left figure, the right vertex in row 3, (6, 3), has to be pushed to (3, 3) to ensure that the wire is routable. The wire runs into a dead end when (6, 3) is chosen because the wire has to be routed horizontally first before it can go down to the next row in the original program. With the top cell blocked by the grid's left boundary, this route is impossible to connect. This is why having two options for the mid-point has the potential to waste less area. The left and right vertices in each row do not necessarily need to be pushed towards the middle for routability because a different mid-point can cause the vertex to be routable.

The choice between the two mid-points is actually straightforward: the one that is routable is chosen. For example, in Figure 5.2, mid-point (6, 2) is not chosen because it is outside the grid's boundary. If both mid-points are inside

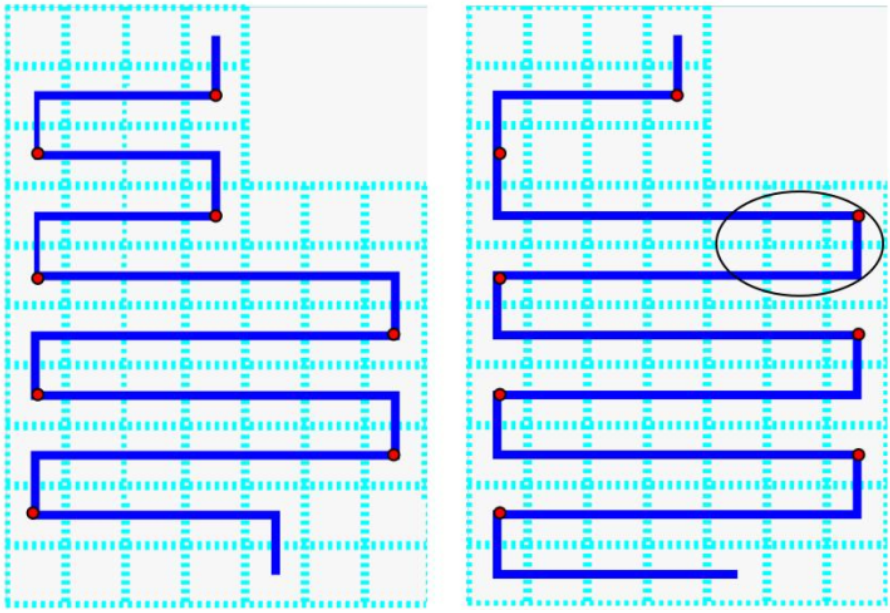

Figure 5.2: The comparison of routable and unroutable last row with vertices shown.

the boundary, the one that is above the current cell is chosen to match closer with the result in the original program. The reason is to avoid discrepancy during length calculation, which will be discussed later. If both mid-points are outside the boundary, there have to be two mid-points to connect the two cells. Two examples are given in Figure 5.3. It can be seen that both mid-points, (0, 8) and (5, 9), are beyond the grid's boundary in the left figure for the cells (5, 8) and (0, 9). One mid-point is chosen in each row. The x coordinate is that of the first cell to which the net can go downwards without being blocked by the previous net or the grid's boundary. The two mid-points are (4, 8) and (4, 9) for the left net in Figure 5.3. Another example is shown in the right net. The two mid-points, (3, 2) and (3, 3), are chosen between the two cells (0, 2) and (4, 3).

5.2 Problems Caused by Having Two Middle Points

Depending on which middle point the program chooses to route, nets can be potentially trapped in the vertex that is routable only when a specific mid-point is chosen. An example is given in Figure 5.4, which shows only one net

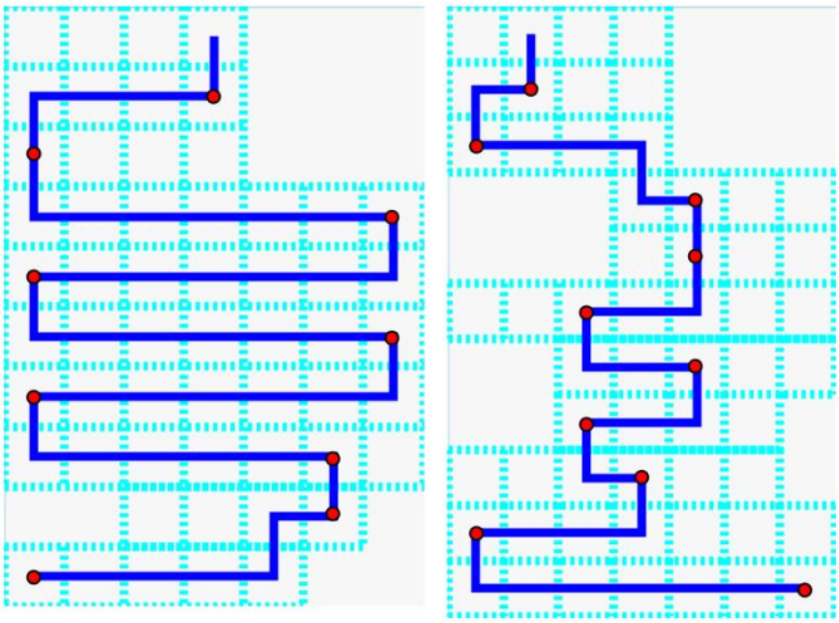

Figure 5.3: Two mid-points are used for connecting two vertices.

from the original solution. The vertex (13, 8) cannot be routed because the net will run into a dead end. If the mid-point (13, 7), was chosen instead of (8, 8), the vertex becomes routable in that case. Since the mid-point is determined after each vertex is chosen during routing, the vertex needs to be dynamically adjusted to avoid traps like (13, 8) in the routing process.

In the original program, such traps can be easily avoided because there is only one way to choose the mid-point. As a result, the traps can be determined and avoided during the assignment of left and right vertices in each row. The path that the original program would produce is shown on top of the new solution in Figure 5.5. A trap exists at (13, 6) for the original program. After a trap is determined, the vertex that causes the trap needs to be moved to a new location. If it is a right vertex, the new location is $(R_i - 1, y)$ where y is the y coordinate of the trap vertex and R_i is the right boundary in row y-1. If it is a left vertex, the new vertex is $(L_i + 1, y)$ where y is again the y coordinate of the trap vertex and L_i is the left boundary in row y-1. The route produced by the original program is 46, which is 6 units shorter than the result of the new program in Figure 5.4.

However, this old method does not work if trap cells cannot be determined prior to the process of choosing the mid-point. A trap cell can only be

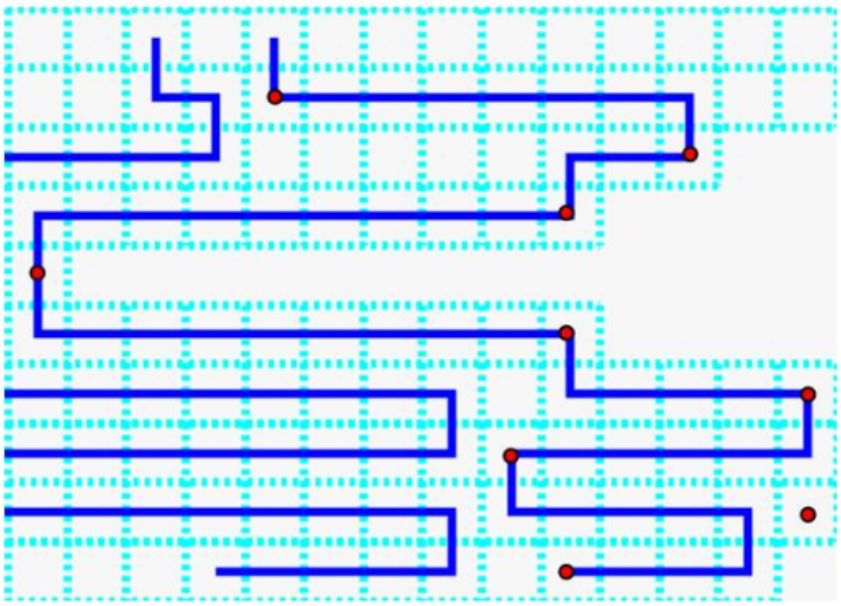

Figure 5.4: An example of an unroutable vertex during routing.

determined during the process with the new option of the mid-point for the program. Therefore, when the program tries to find the mid-point to connect two vertices, it checks whether the previous vertex is a trap cell. If any direction except up is routable, the vertex is not a trap. If it is a trap, the net is in a dead end and the program immediately aborts the routing process. The program pushes the trap vertex on the *trap* vector and clears the *route* vector and the *vertices* vector. Then, the routing process goes back to the stage that determines the vertices in each row. Now, the program has the information of which cells will become trap cells with the current boundary. Therefore, if the vertex is a left vertex, it is adjusted to $(L_i^N+1,\ y)$, where L_i^N is the new left boundary in row $y+1$. If it is a right vertex, it is adjusted to $(R_i^N - 1,\ y)$, where R_i^N is the new right boundary in row $y+1$. The new high level pseudo code for *find_route* is shown below:

```
//the minimum labels have been assigned prior to find_route
procedure find_route:
    determine the vertices in each row
    complete the route vector by either pushing an adjacent cell or a vertex
    determine the mid points for every non-adjacent cell in the route vector
    if encounter a trap vertex
```

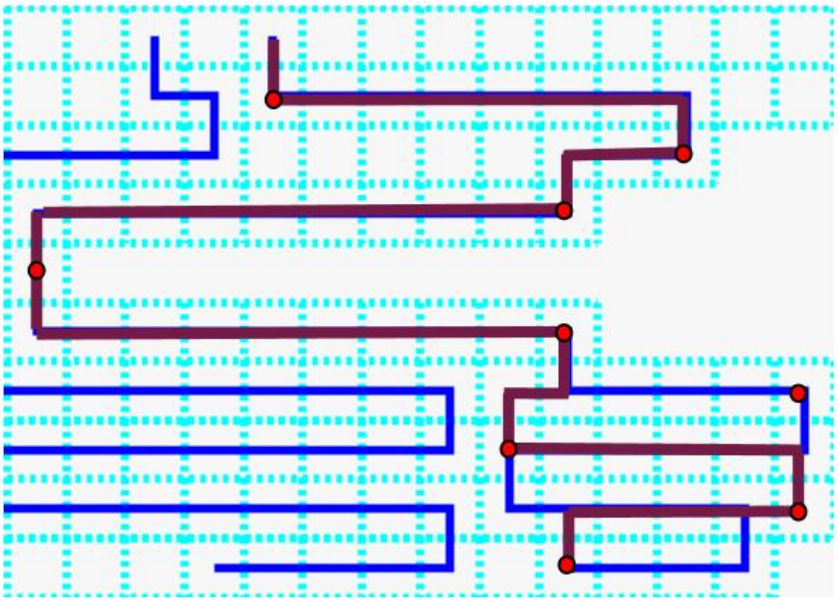

Figure 5.5: An example of the routing if the other mid-point is chosen.

clear *route* vector, push the trap vertex to the *trap* vector
go back to determine the vertices in each row again

The net can have extra grid cells for snaking with a different mid-point that connects the nonadjacent grid cells as in Figure 5.2 and Fig 5.5. Unfortunately, this longer wire cannot finish routing sometimes due to a trap vertex. If it encounters a trap, the vertex needs to be moved to make the net routable, and it is possible that the wire cannot gain any length at all after the vertex is moved. An example is given in Figure 5.6.

The first attempt to connect all the nonadjacent vertices is shown in the left graph. In row 3, the right vertex is chosen because the right vertex has a longer edge than the left vertex. However, this choice will make the vertex (0, 8) a trap vertex, since the program does not know the grid cell below (0, 8) is not routable; it only knows that when it tries to connect (0, 8) and (6, 9). Since the trap is a left vertex, it is moved to (5, 8). With this new vertex, the program tries to find the route that produces the maximum-length again. The result after the program encounters no trap vertex is shown at the right for the same net. It can be seen that the choice of going to the right vertex in row 3 actually produces a result that is shorter than going to the left vertex.

Therefore, the program chooses the left vertex in row 3 instead of the right vertex to get the maximum-length for the net.

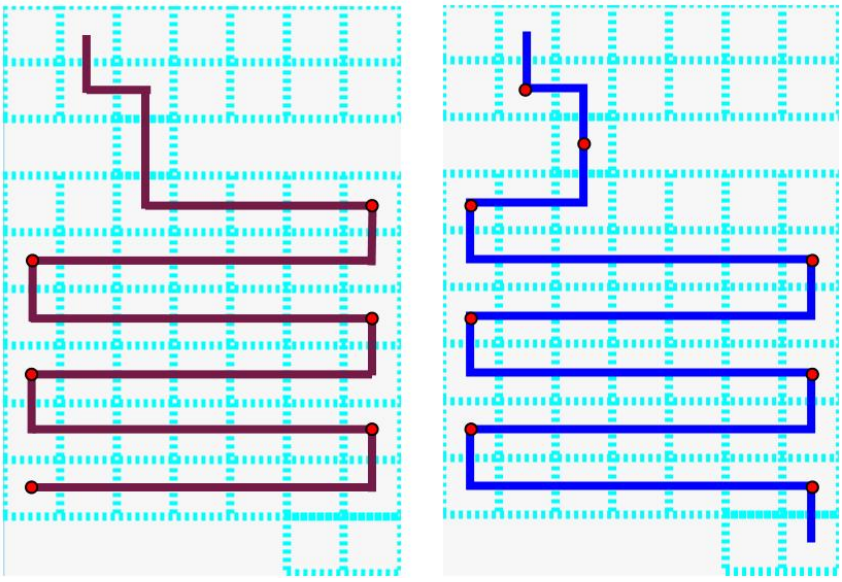

Figure 5.6: An example of how an original attempt to route can result into a dead end.

Since the program calculates the maximum-length with the assumption that there are no trap vertices, the maximum-length needs to be recalculated again with the new vertices that replaced the traps. In Figure 5.6, the maximum-length is 50 before the trap vertex, (0, 8), needs to be moved to (5, 8). After the vertex is moved, the new maximum-length is only 42. If this new maximum-length, which is shorter than the original one, is below the minimum-length constraint of the net, not enough grid cells are allocated for the current net. In that case, the right boundary of the net (R_i) needs to be flipped more towards the right to allocate more cells for snaking. Remember that R_i is only now used to determine the right vertex in each row for snaking. Therefore, flipping R_i to the next column is equivalent to moving the right vertices to the next column, which allocates more cells for snaking. The flipping process continues until the maximum-length, which is determined after all the vertices are routable with the mid-points chosen for each nonadjacent cell, is greater than the minimum-length constraint.

CHAPTER 6

MEETING THE MAXIMUM-LENGTH BOUND

6.1 Introducing Diagonal Wires

In the original program, maximum-length bound is only checked to allocate individual net's right and left boundaries, L_i and R_i. The program assumes that all nets will satisfy the maximum-length bound if wire snaking is not required. Therefore, as long as the nets stay inside L_i and R_i, the resulting path length cannot exceed the maximum-length bound. However, if the minimum path does not meet the maximum-length requirement, the only solution is to either change the terminal pins' position or increase the maximum-length bound.

Since the wires always route in either horizontal or vertical direction, the minimum path is always the Manhattan distance between the two terminal pins. The formula to calculate the Manhattan distance is $|x^T - x^B| + h$, where x^T is the x coordinate of the top terminal pin, x^B is the x coordinate of the bottom terminal pin, and h is the height of the grid. Since the minimum distance between two points is a straight line, the wire length can be less than the Manhattan distance if the wire can be routed in directions other than horizontal and vertical. For this reason, diagonal wire should be used during routing.

If diagonal wire is added in the original program, it will raise many problems because the original program was not designed to incorporate diagonal wire. Many fundamental changes will need to be made to add the diagonal wire. Therefore, we wrote a separate program (*diagonal*) that uses diagonal wire to route in a smaller region inside the grid's boundary. After the *diagonal* program finishes routing, it modifies the original input file, along with *lObstacle* and *rObstacle*, to exclude the region that has been assigned for the diagonal region already. Then, it calls the *route* program to finish the

routing in the remaining region. At the end, the *diagonal* program draws the routing result in the diagonal region and the *route* program draws in the remaining area. The two graphs are in the same grid and the combined graph corresponds to the original input and boundary information. Therefore, the *diagonal* program acts like a preprocessing program for the input file before the *route* program. The *route* program does not know that the *diagonal* program exists, so it is completely transparent.

6.2 The Number of Cells of Diagonal Wire

The *diagonal* program first reads the input file and determines the minimum-path distance for each net by calculating the Manhattan distance. If each minimum-path distance satisfies each maximum-length constraint, the program does not need to use diagonal wire. In that case, the program does not modify the files and simply calls the *route* program, never using the *diagonal*. If one of the net's min-path distances is less than the maximum-length bound, the program proceeds to calculate the number of cells for diagonal wire that will satisfy the maximum-length bound.

To determine the length of diagonal wire, the program calculates the difference between the maximum-length bound and min-path distance. The result (l_{diff}) is the wire length that has to be saved to meet the maximum-length bound. The program then calculates the height (h_d) of the diagonal wire to save l_{diff}. The length saved by diagonal wire in one cell is the length difference between the diagonal and the Manhattan distance, which is $2 - \sqrt{2}$. Therefore, the height of the cells occupied by diagonal wire for a net is determined by the formula: $h_d = \lceil l_{diff}/(2 - \sqrt{2}) \rceil$. Since the program has to satisfy the maximum-length bound for all the nets, h_d is calculated for each wire, and the maximum h_d is chosen to be the height of the diagonal region.

The new minimum-path distance with using diagonal wires can also be determined. The diagonal wire is limited by the coordinates of the terminal pins. For example, if the top terminal pin is at (5, 0) and the bottom terminal pin is at (10, 10), the diagonal wire cannot go beyond the rectangle formed by four vertices, (5, 0), (10, 0), (10, 10), and (5, 10). If the diagonal wire goes beyond this rectangle, it starts to deviate from the destination terminal pin, which does not save wire length anymore. The extra length is $\sqrt{2}$, since

the deviated wire uses $1 + \sqrt{2}$ length, while the wire that directly goes up uses 1. Therefore, the maximum height used by the diagonal region (h_d^M) is $min(|x^T - x^B|, h)$. The minimum distance of the net (D_m) is calculated by the formula $D_m = \sqrt{2} \times h_d^M + max(|x^T - x^B|, h) - h_d^M$. Therefore, the program checks the minimum path for all nets and immediately aborts if a min-path fails to satisfy the max-length bound.

6.3 Determining the Region for Diagonal Wire

After the h_d that satisfies all nets has been chosen, the next step is to find a region inside the grid to place diagonal wires for each net. Since wires do not cross the same row twice, the free cells that are not used in the diagonal region cannot be used for the *route* program. Therefore, the diagonal region is determined by cutting horizontally in multiple rows. The rows that are not assigned for the diagonal region are later used for the *route* program.

The program favors putting the diagonal region at the bottom or at the top because it has to call the *route* program only once. All remaining grid cells after the diagonal program picks its region are used for the route program. For example, if the rows from $h - h_d$ to h are chosen for diagonal wire, the rows from 0 to $h - h_d$ will be used for the route program. However, the pin positions for the diagonal wires are stuck to the pin positions at the top or bottom. They cannot be moved closer or farther apart.

The program tries using the bottom rows first, which are from $h - h_d$ to h. Then, it determines if all the wires can fit into the grid by examining the first and the last nets' diagonal wires. For each row from $h - h_d$ to h, the diagonal wire's x coordinate is checked against the left boundary and the right boundary of the grid. If the wires cannot fit, the diagonal wire region is moved to the top.

If the diagonal wires do not exceed the boundary, the program draws the diagonal wires in the direction towards the top terminal pins from the bottom. Then, it updates the input file with the new bottom pin positions based on the cells at which the diagonal wires terminate. The minimum- and maximum-length bounds are also updated by subtracting the length of the diagonal wire. The height of the grid is updated as well. Finally, the program calls the *route* program to finish the routing in the top region. An example is

given in Figure 6.1.

If the top rows, which are from 0 to h_d, are chosen, the *diagonal* program checks if all the wires fit inside the grid, using the same method that was used to check the bottom rows. If a diagonal wire touches the boundary, the middle region has to be used. If not, the program draws the diagonal wires in the direction towards the bottom terminal pins from the top. Then, it updates the input file with the new top pin positions based on which cells the diagonal wires route to. The minimum- and maximum-length bounds are also updated with the height of the grid. Finally, the program calls the *route* program to finish the routing in the bottom rows. An example is given in Figure 6.2.

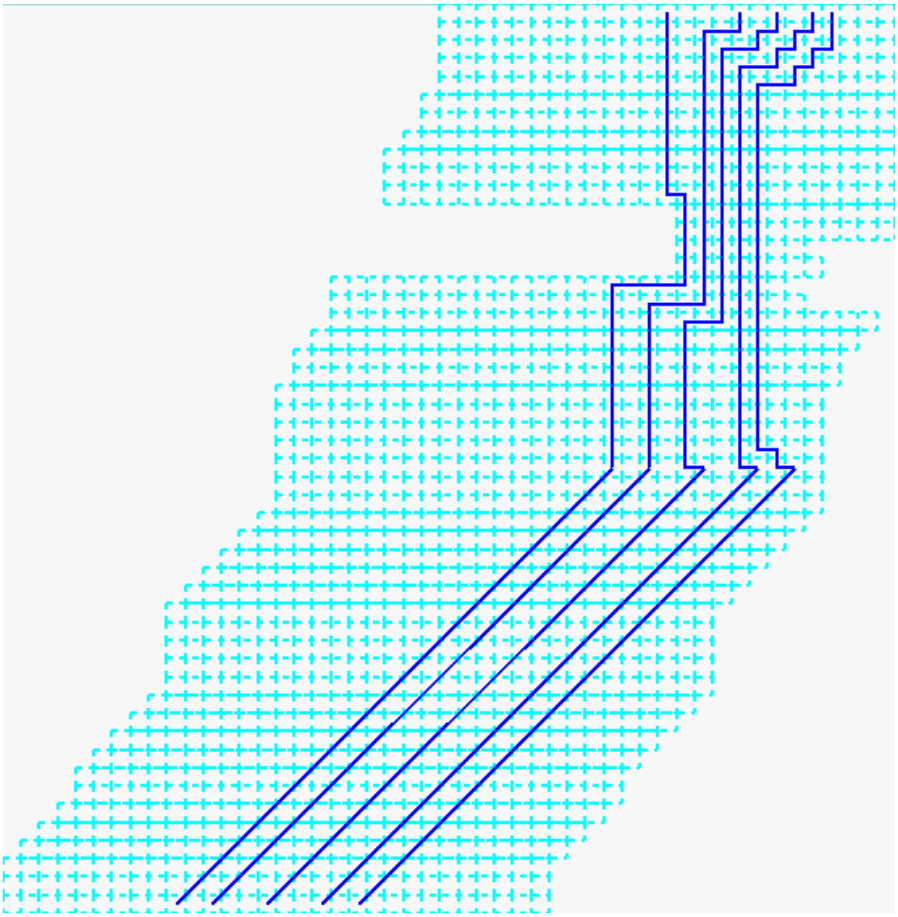

Figure 6.1: An example of diagonal wires in the bottom rows to satisfy max-bound.

If both the top and bottom regions fail, the diagonal region has to be in the middle. In this case, the positions of the starting points of the diagonal

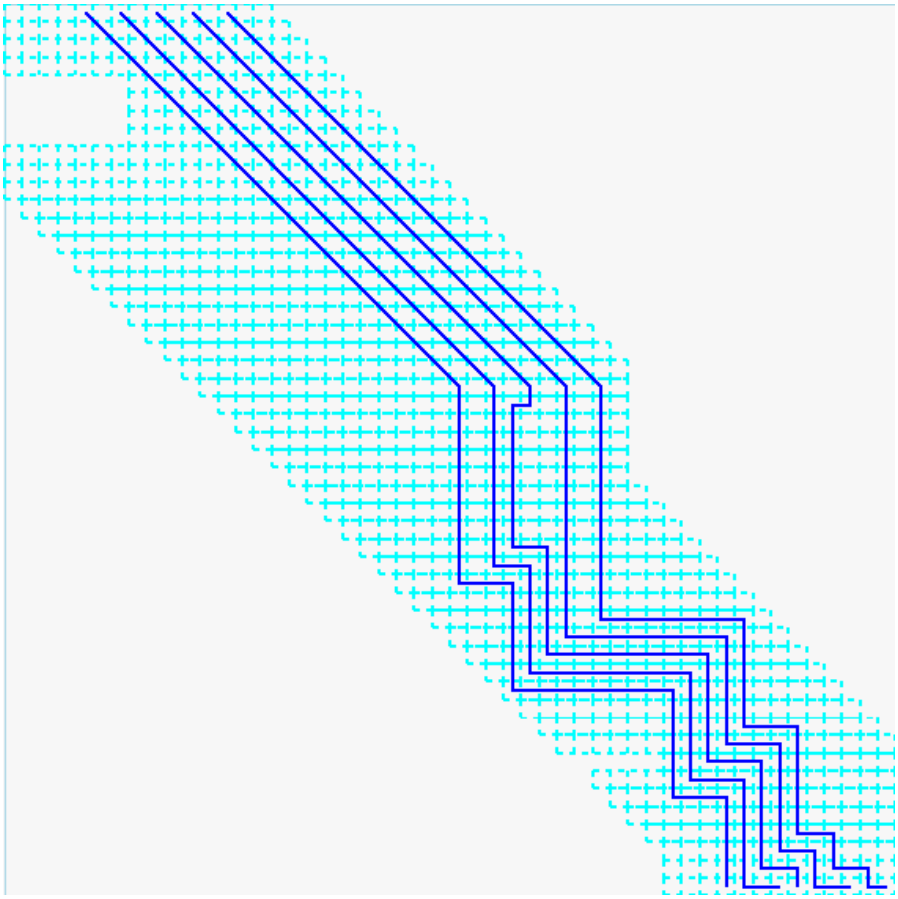

Figure 6.2: An example of diagonal wires in the top rows to satisfy max-bound.

wires do not have to be the same as the terminal pin positions. They can be moved at the edge of the diagonal region. Therefore, the pins can potentially be moved closer together so the resulting wires look more like a bundle. The program arbitrarily chooses the bottom of the middle region in row $h/2+h_d/2$ to make the diagonal region exactly in the middle. An example of diagonal region in the middle is shown in Figure 6.3.

The pins are chosen to let the diagonal wire for each net be inside the rectangle formed by the top and bottom terminal pins. Therefore, the optimal position for each net will be $(x^B,\ h/2 + h_d/2)$, where x^B is the x coordinate of the bottom terminal pin. The program checks the first and last diagonal regions again to see if they touch the boundary. If one of them touches the boundary, the program tries to avoid the boundary by moving the pin positions left or right. If the pin positions cannot be moved to fit the rectangle,

the program switches the bottom of the diagonal region to upper or lower row and tries the entire process again. The program continues searching a position for the region until a diagonal region can fit in the grid where all maximum-length bounds are satisfied.

If a diagonal region is decided, the grid is split into three different sections, and the *diagonal* program has to call the *route* program twice to finish the routing in the top and bottom regions. Different input files need to be produced with new top and bottom pin positions and new length bounds. The new max-length bound is determined based on the min-path distance of the target region. The min-path distances for the top and bottom regions are added together to form a total min-path distance. The extra wire length is defined to be $(oldmax\text{-}bound - \sqrt{2} \times h_d - totalmin\text{-}path)/2$. This extra wire length is then split evenly between the top and bottom regions to add to the min-path distance to form the new maximum-length bound. Therefore, the region with higher min-path distance gets a larger max-length bound. The new minimum-length bound is calculated by subtracting $\sqrt{2} \times h_d$ from the old minimum-length bound and dividing by 2. The three different regions are combined together for a single solution after they finish the routing.

6.4 Satisfying the Maximum-Length Bound with the Diagonal Region

Since the diagonal wires' lengths have to be uniform for all nets, there can be cases where the wire length gets shorter than the minimum-length bound if minimum-path is followed. However, this problem can be easily resolved because the *route* program can lengthen the wire by snaking in its region. On the other hand, the cases where the maximum-length constraint cannot be satisfied are harder to solve. For some nets, the diagonal wire becomes so long that it has to exceed the rectangle formed by the top and bottom terminal pins. After it exceeds the rectangle, the diagonal wire actually makes the wire longer than necessary, which can cause it to exceed the maximum-length bound. Unfortunately, there seems to be no easy solution to this problem other than using variable diagonal wire lengths for different nets. As a result, each net can have different length of diagonal wire to satisfy the maximum-length bound. However, this raises the old challenge of routing horizontal

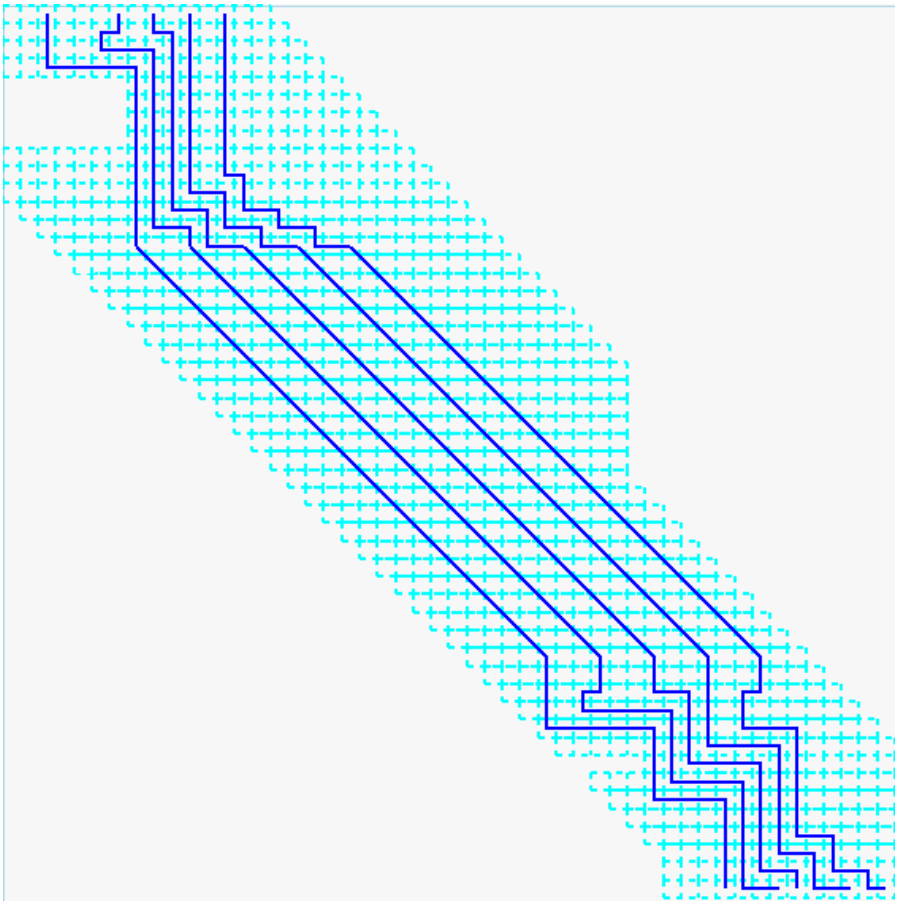

Figure 6.3: An example of diagonal wires in the middle to satisfy max-bound.

and vertical wires with diagonal wires in the same region.

6.5 Desired Length for Each Net

When a wire length meets the minimum- and maximum-length bounds, it is not set to be close to the middle of the two values. Wire length is always close to the maximum-length bound because extra wire length is used to route the nets closer together and leave more grid cells for the rest of the nets. However, this is not always the desired situation, so a desired value for the wire can be specified by the user in the input file after the maximum-length value. As a result, the wire length will be routed closer to the desired length instead of the maximum-length bound. Therefore, the user has the

freedom to change the wire length when the minimum- and maximum-length bounds are satisfied.

CHAPTER 7

COMPARISON

In this chapter, two examples are presented to show the differences between the original program and the new program.

In this example, there are eight nets to route, and the maximum-length bound can be satisfied with the Manhattan distance for each net. Figure 7.1 is the result from the original program. The last net is away from the rest of the nets. In addition, some nets cannot satisfy the minimum-length bound. For example, the first net has minimum-length bound of 80, but the resulting wire length is only 75. Figure 7.2 is the result of the newer program. The first net can satisfy the minimum-length bound now because it has more grid cells for snaking. The nets in the original program do not have as many grid cells for snaking because they were preallocated for each net before the routing process. The last net is also routed closer to previous nets in the result of the newer version. Figure 7.3 is the result of the newer version with the addition of left and right boundaries. The nets are routed tighter together to route around the left and right boundaries.

In the next example, all the nets except for the second one cannot satisfy the maximum-length bound, which is 80. The shortest path is the Manhattan distance in the original program, and the user has to adjust the pin position or the maximum-length bound. The result is shown in Figure 7.4. With the addition of the diagonal wires in the newer version, the maximum-length bound can be satisfied by using the diagonal region in the middle. All nets are below the maximum-length bound in this case. The result is shown in Figure 7.5

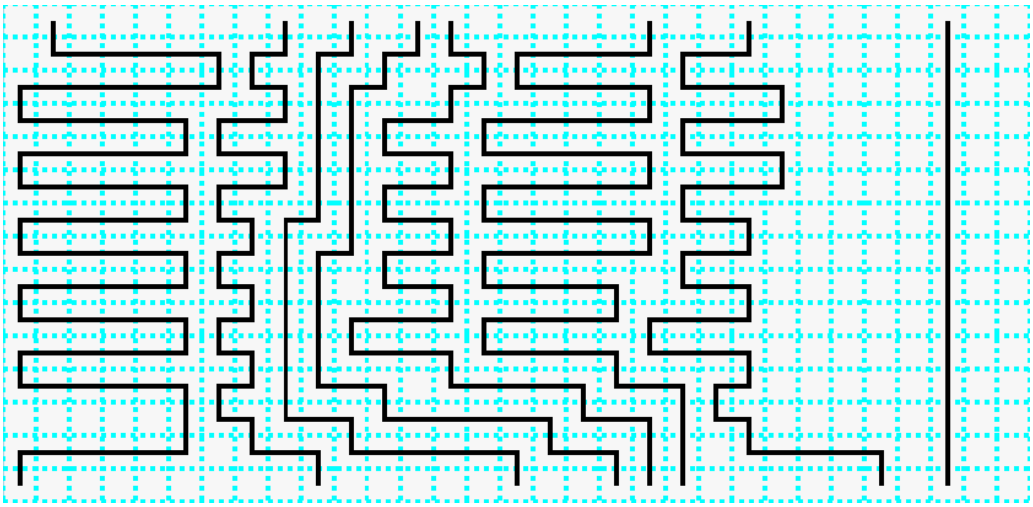

Figure 7.1: The result of the original program.

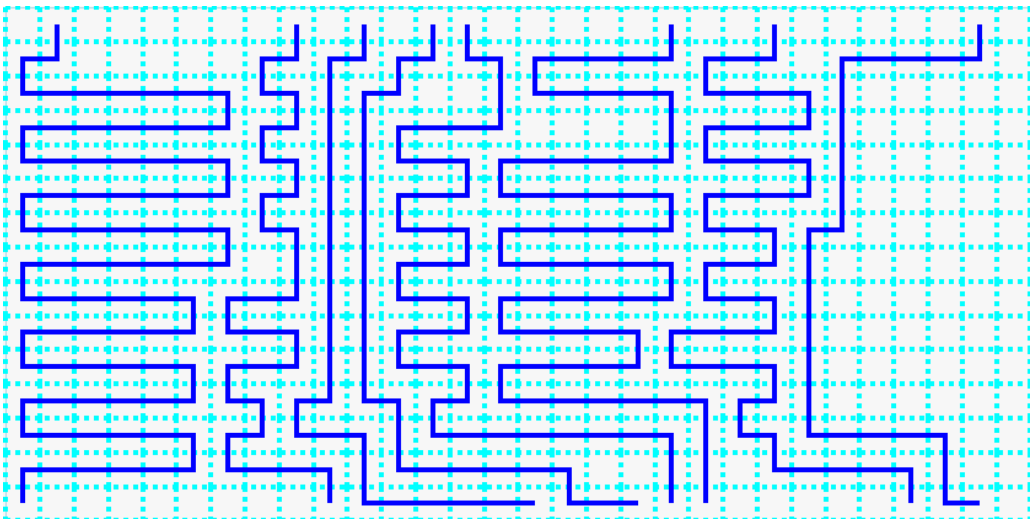

Figure 7.2: The result of the new program with no boundary.

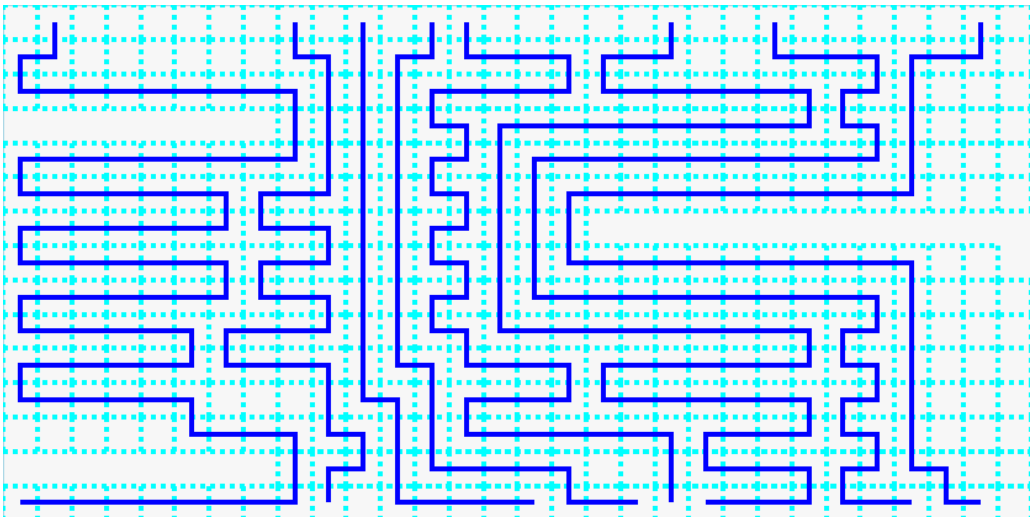

Figure 7.3: The result of the new program with boundary.

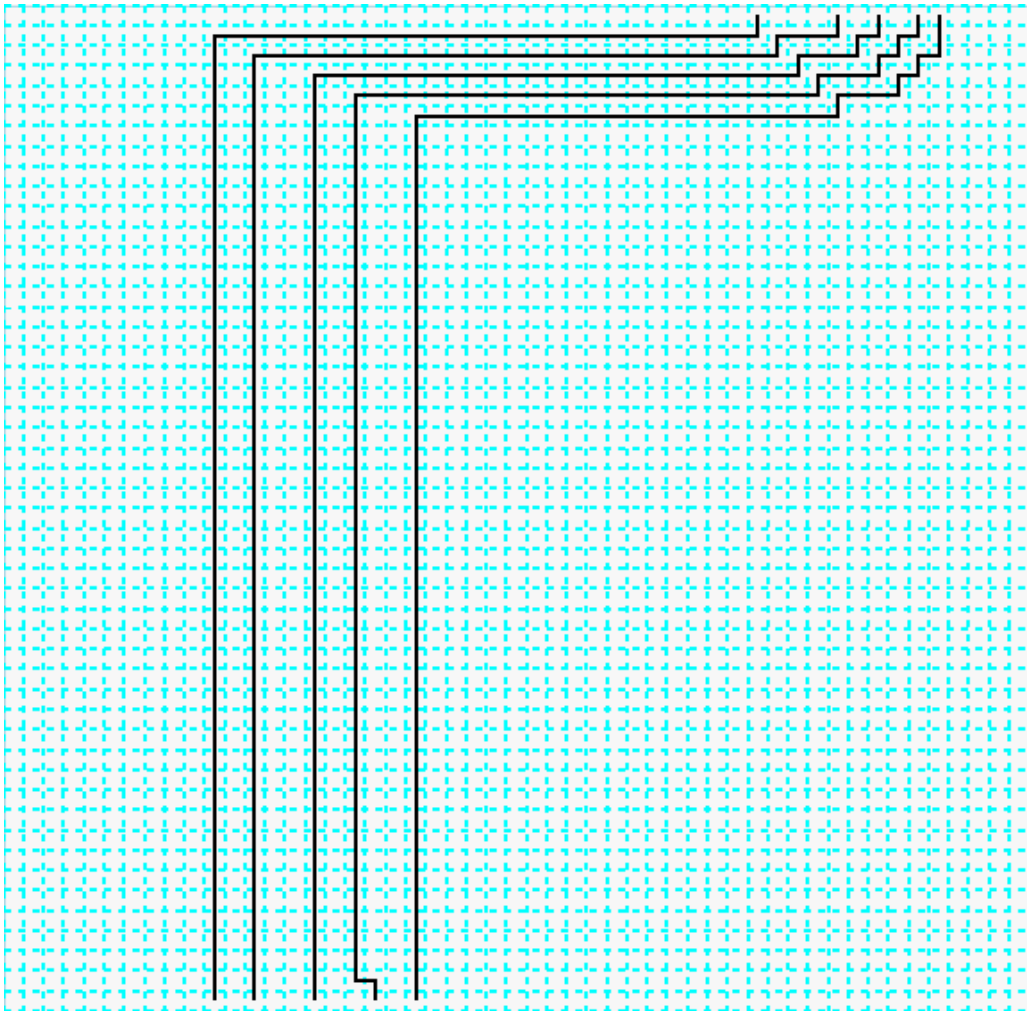

Figure 7.4: The result of the original program with no boundary and no diagonal wires.

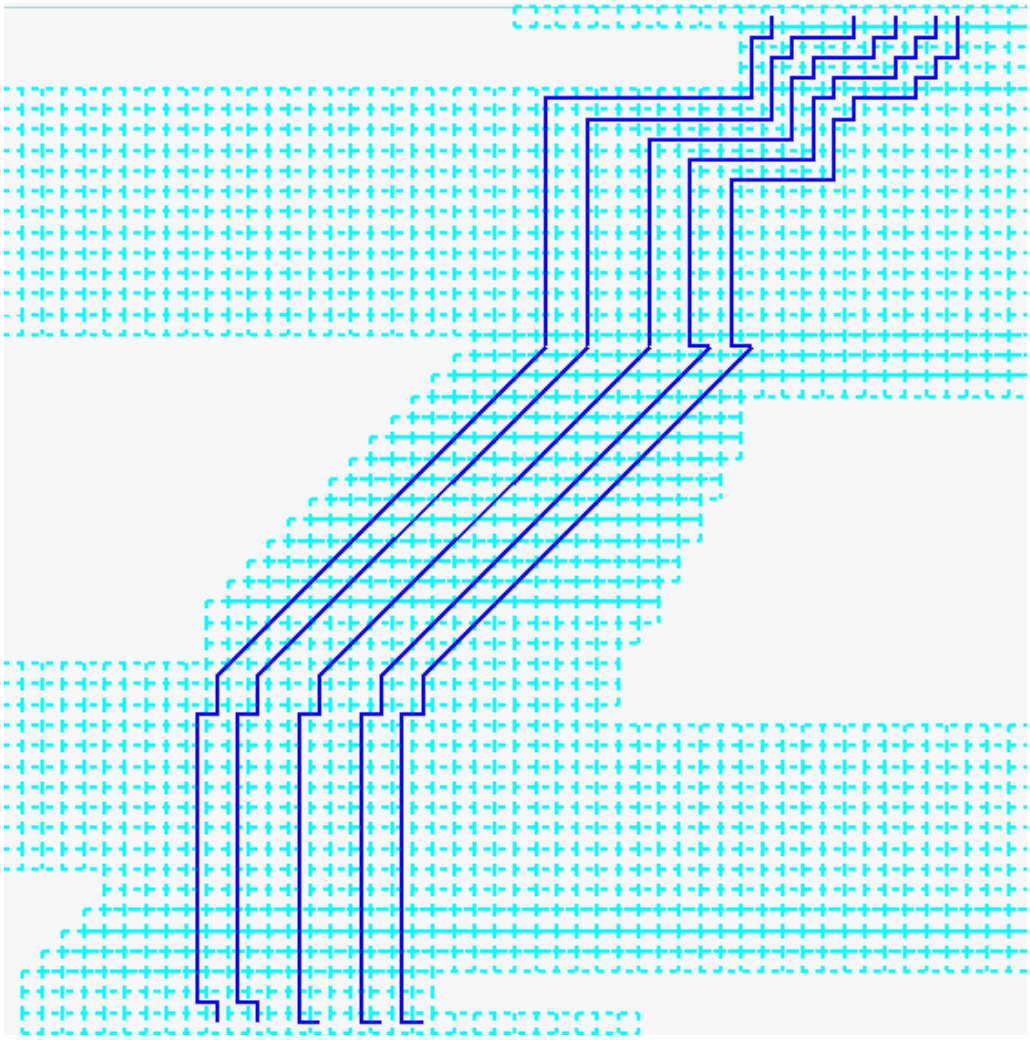

Figure 7.5: The result of the new program with boundary and diagonal wires.

CHAPTER 8

CONCLUSION

8.1 Summary

Ozdal and Wong's program is an automatic single-layer bus routing program that matches wire lengths on high-speed boards. We try to improve the program so it becomes more general to accommodate different input nets. The program now does not have to route inside a uniform grid. The user can specify left and right boundaries to model the obstacles that might be encountered during PCB routing. Each individual net's boundary does not have to be monotonic anymore. Most of the cells inside the grid's boundary are available for the current net as long as they are not essential for the remaining nets to connect the two terminal pins. Therefore, the earlier nets can more likely finish routing than the later nets. The nets that have finished routing are drawn in the solution grid right away. As a result, the half-finished routing result is still available for the user when a net fails to satisfy the min-length or max-length bound.

During the actual routing for the net, the program makes an effort to have less bending in the path by giving priority to the previous direction. However, those routes can potentially waste cells, so the program gives the user the capability to enforce a cap on the directions that waste cells for the remaining nets. The program makes more cells available by making the last row of the grid routable for each net. As a result, more nets can meet the minimum-length constraint with tighter boundary.

The program can also use the diagonal wires in cases where the maximum-length bound cannot be satisfied by the minimum-path length. A second program, *diagonal*, is used to preprocess the input file before the *route* program. It checks the minimum-path distance for each net against the maximum-length bound and routes part of the grid in diagonal wire in case the min-path

distance is less than the max-length bound. The diagonal region is chosen to be either at the bottom, at the top, or in the middle, depending on which region can contain the diagonal wires. The diagonal wire length has to be uniform for every wire to avoid the challenge of routing horizontal, vertical, and diagonal wires in the same region. After all the nets finish routing, the length of each net is automatically displayed in the terminal for the user.

With all these modifications, the single-layer bus routing program can be used without the original limitations of the nets input and environment.

8.2 Future Work

There are some improvements that can be made to this program. The *diagonal* program assumes that all nets use diagonal wires in the same direction. In reality, the nets can be in two different directions, 45° left and 45° right. As a result, they will need the program to draw the diagonal wires in these two directions in the same diagonal region. Furthermore, there are cases where it is impossible to satisfy the maximum-length bound for all nets using diagonal wire of uniform length. Therefore, different diagonal wire lengths for different requirements can potentially solve this problem. However, the problem of routing wires of different lengths can be challenging. In addition, making the first row routable by the nets provides even more grid cells during routing. Lastly, the program should make a more intelligent guess of which mid-point to choose during routing, rather than taking a trial-and-error approach, which wastes a lot of runtime.

REFERENCES

[1] L. W. Ritchey, "Busses: What are they and how do they work?" *Printed Circuit Design Mag.*, Dec. 2000.

[2] M. Ozdal and M. Wong, "Length matching routing for high-speed printed circuit boards," in *Proc. IEEE Int. Conf Computer-Aided Design*, San Jose, CA, Nov. 2003, pp. 394–400.

[3] M. Ozdal and M. Wong, "Algorithmic study of single-layer bus routing for high-speed boards," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, pp. 490–503, Mar. 2006.

[4] C. Hsu, "General river routing algorithm," in *Proc. 20th Design Automation Conf.*, Miami, FL, June 1983, pp. 578–582.

[5] R. Y. Pinter, "On routing two-point nets across a channel," in *Proc. 19th Design Automation Conf.*, Las Vegas, NV, June 1982, pp. 894–902.

[6] H. Zhou and M. Wong, "Optimal river routing with crosstalk constraints," *ACM Trransact. Des. Automat. Electron. Syst.*, vol. 3, no. 3, pp. 496–514, 1998.

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

[8] S. W. Hur, A. Jagannathan, and J. Lillis, "Timing-driven maze routing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits and Syst.*, vol. 19, pp. 234–242, Feb. 2000.