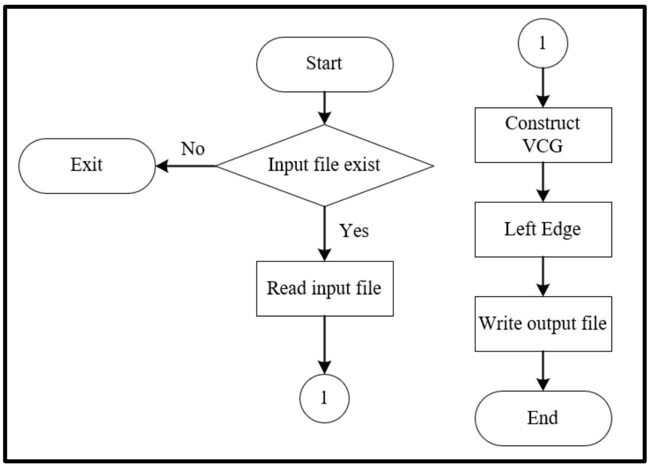
A. Readme

I. Flow Chart

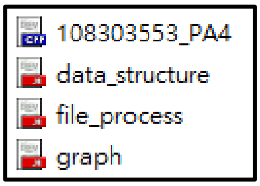


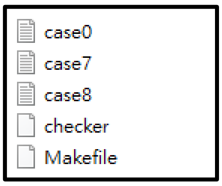
II. Compile and Execute

First, use "mkdir" command to create a directory named "PA4", then use "cd" command to enter this directory.

```
[s108303553@eda359_forclass ~]$ mkdir PA4
[s108303553@eda359_forclass ~]$ cd PA4
```

Next, put following files into this directory. Including cpp file, three header files, Makefile, checker and three testcases. The test data on the workstation must be in **Unix format**, otherwise the program will output incorrect results.





Then, use "chmod 700 checker" command to modify checker's permission so that we can use checker to check the answer. After that, use "make all" command to execute the cpp file automatically. We can find that "108303553_PA4.o" and "exe" are created after this command.

```
[s108303553@eda359_forclass ~/PA4]$ chmod 700 checker
[s108303553@eda359_forclass ~/PA4]$ make all
[s108303553@eda359_forclass ~/PA4]$ ls
108303553_PA4.cpp  case0.txt  case8.txt  data_structure.h  file_process.h  Makefile
108303553_PA4.o    case7.txt  checker    exe                graph.h
```

Now, we can use "make run input=input_file output=output_file" to generate out file. Later, use "make clean" command to remove "108303553_PA4.o" and "exe".

```
[s108303553@eda359_forclass ~/PA4]$ make run input=case0.txt output=out0.txt
[s108303553@eda359_forclass ~/PA4]$ make run input=case7.txt output=out7.txt
[s108303553@eda359_forclass ~/PA4]$ make run input=case8.txt output=out8.txt
[s108303553@eda359_forclass ~/PA4]$ make clean
```

Finally, use "./checker out.txt case.txt" to check whether the output file is correct or not.

```
[s108303553@eda359_forclass ~/PA4]$ ./checker out0.txt case0.txt
```

B. Completion

```
---------- Status Report ----------
track count: 3
All signals are connected successfully.
-----------------------------------
```

case0

```
---------- Status Report ----------
track count: 3
All signals are connected successfully.
-----------------------------------
```

case7

```
---------- Status Report ----------
track count: 4
All signals are connected successfully.
-----------------------------------
```

case8

We can see that all output files pass the check. The following table shows the track count of each testcase.

Testcase	case0	case7	case8
Track Count	3	3	4

C. Data Structure and Algorithm

I. Data Structure

```
//------- define data structure of graph node -------
namespace data_structure
{
    // horizontal edges in left edge algorithm
    struct Edge_Link
    {
        int start, end;          // start <--------> end
        int id;                  // pin id of this net
        int track;               // the track to which the edge belongs
        bool type;               // 0: space edge, 1: solid edge
        struct Edge_Link *next;  // next edge
    };

    typedef Edge_Link* Edgeptr; // name Edge_Link* as Edgeptr

    // data structure of VCG
    struct Node
    {
        int tail;           // tail id
        int head;           // head id
        bool state;         // shows that this node is teaversed or not
        struct Node *psor;  // presuccessor
        struct Node *ssor;  // successor
    };

    typedef Node* Nodeptr; // name Node* as Nodeptr

    struct VCG
    {
        Nodeptr sptr; // successor
        Nodeptr pptr; // presuccessor
        vector <Edgeptr> eptrs; // horizontal edges
    };

    struct Pin_Location
    {
        bool top; // 1: top pin, 0: bottom pin
        int col;  // the position (column) of this pin
    };

};
```

The first one is "Edge_Link", which includes an edge’s terminal position, ID, track (height), type (space edge / solid edge) and the pointer point to next edge. The second one is "Node", which is the data structure of VCG (vertical constrain graph). The third one is "VCG", which includes each pin's edges and pointer point to successor and pre-successor. The last one is " Pin_Location", which can store each pin’s position.

II. Main Module

```
int main(int argc, char *argv[])
{
    vector < vector <string> > input; // string segmentation from input file
    vector <VCG> vcg;                 // vertical constrain graph
    int track_max = 0;                // the number of tracks

    //-------  read input file and store the data into a 2D string vector  --------
    File_Process file;
    file.in_file(input, argv[1]); // read input file

    //--------------------  use left edge algorithm to route  --------------------
    Graph graph(input);
    graph.left_edge(vcg, track_max);

    //---------------------------  write output file  ----------------------------
    file.out_file(vcg, track_max, argv[2]);
}
```

In main module, I declared three variables:

1. input: A two-dimensional string vector that stores each word in input file.
2. vcg: The vertical constrain graph.
3. track_max: The number of tracks after routing.

First, invoke the "in_file" class method from the "File_Process" class to read the input file and return its contents. Then, use the "left_edge" method from the "Graph" class to route the channel and return the number of tracks. Finally, write the result to output file though "out_file" method.

III. File Process

```
//------- read input file and write output file -------
class File_Process
{
private:
    // transfer sentence to substrings
    vector <string> sentence_to_substrings(string input) // Ex: save "Hello World" as "Hello" and "World"
    {

    // transfer string to integer number
    int StoI(string num)
    {

    vector < vector <string> > input_segmentation;

public:
    // read input file
    void in_file(vector < vector <string> > &lvs, char *argv)
    {

    // write output file
    void out_file(vector <VCG> vcg, int track_max, char *argv)
    {
};
```

1.Private Functions

(1) sentence_to_substrings: Split a sentence by spaces and store them into a vector.

(2) StoI: This function can convert a string to an integer data type.

2.Public Functions

(1) in_file: The function can read input file and return the contents.

(2) out_file: It is capable of writing the results to an output file in a specified format.

IV. Graph

```
//------------------  create graph  -----------------
class Graph
{
private:
    vector < vector <string> > input; // input data from input file
    int col_size;                     // column size of pins from input file

    // transfer string to integer number
    int StoI(string str)
    {

    // create a new node in VCG
    Nodeptr NewNode(int Tail, int Head)
    {

    // create a new edge
    Edgeptr NewEdge(int start, int end, int id, bool type)
    {

    // use for finding all horizontal edges in these nets
    Edgeptr find_edge(int id)
    {

    // compare left x coordinate, return true if edge 1's x is smaller then edge 2, otherwise return false
    static bool compare_x(Edgeptr e1, Edgeptr e2)
    {

    // find all unconstrained edges (doesn't include free edges)
    vector <Edgeptr> unconstrain_edges(vector <VCG> vcg)
    {

    // create a VCG
    void create_VCG(vector <VCG> &vcg)
    {

public:
    // constructor -> initialize private members
    Graph(vector < vector <string> > input_data)
    {

    // the algorithm of routing
    void left_edge(vector <VCG> &vcg, int &track)
    {

};
```

1.Private Function (Member)

(1) input: A vector that stores each word from input file.

(2) col_size: The column size of pins.

(3) StoI: This function can convert a string to an int data type.

(4) NewNode: It can allocate memory to create a new node in VCG.

(5) NewEdge: Similar to "NewNode", it can create a new edge into tracks.

(6) find_edge: Search the longest horizontal edge of pin x.

(7) compare_x: Compare two edges' starting points and return a bool value.

(8) unconstrain_edges: It is responsible for finding edges to place into tracks.

(9) create_VCG: It is designed to build the vertical constrain graph.

2.Public Functions

(1) Graph: Constructor of this class, which initializes its private member.

(2) left_edge: The algorithm of channel routing.

V. Algorithm

My algorithm is same as left-edge algorithm in lecture notes. Therefore, I don't elaborate further. Let's discuss my optimization method, even though I was unable to complete it within the deadline. Take case8 as example:

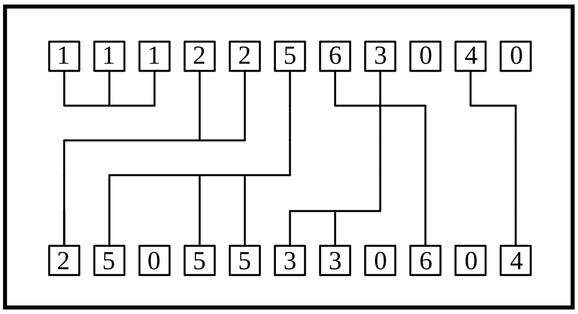


Fig. 1. Left-Edge algorithm result

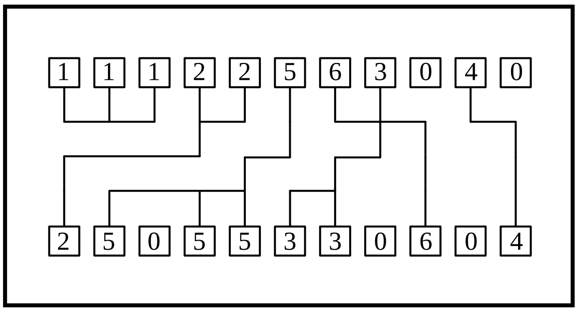


Fig. 2. Optimizing result

Figure 1 is the result after left-edge algorithm, we can find that the total track is three. Figure 2 is my idea of optimizing the routing result. Once we complete the left-edge routing, we can minimize the number of tracks from the result.

First, I use a data structure similar to link list to store each track, as shown in figure 3 below (use Figure 1 as example):

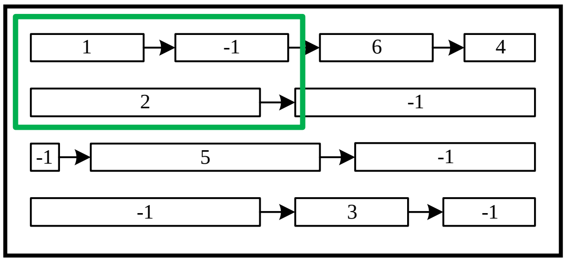


Fig. 3. Edge_Link data structure

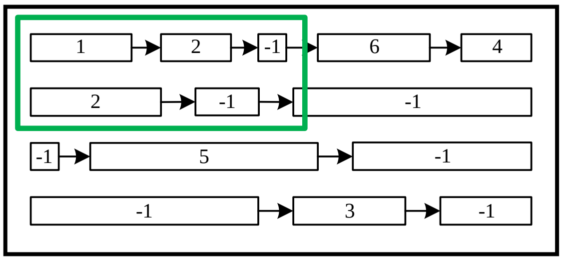


Fig. 4. The structure after spliting

The numbers represent the id of the edge, "-1" means space edge (no edge occupies there). With this data structure, I can traverse all rows to check whether they can be split and can be swapped position or not.

Second, select an adjacent pair of rows. For example, we select the upper two rows, and I find that there is a space edge on the top of edge 2. After checking, edge 2 can be split and swap its position to upper row. As shown in figure 4.

Finally, by repeating these procedures several times, we can obtain the optimal result (Fig. 2). Unfortunately, I encountered some bugs while attempting to output the result to a file, and the deadline is approaching. Perhaps I can deal with this issue after some days...