Design Document for "Multiple-Terminal Path Finding by Column Generation Algorithm"

YU Jiping, 2015011265

Contents

1 Problem Specification

1.1 Problem Description

Given a rectangular board sized $N \times M$ with obstacles and P sets of terminals, each containing 2 or more points.

It is expected to connect some sets of terminals (with Steiner trees allowed). No two lines can cross each other, and no line cross obstacles.

The goal is first to maximize the number of connected terminal sets, and then to minimize the grids used.

1.2 Input Format

The program (`main.cpp`) reads input from `stdin`.

First, input space-separated integers N and M.

Then, input several terminal set and obstacle specifications.

For an obstacle, input "Obstacle x y" for an obstacle at position (x, y).

For a terminal set, input "Terminal n x_1 y_1 x_2 y_2 ... x_n y_n" for a terminal set $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$.

Since the positions are 0-indexed, the input should meet $0 \leq x < N, 0 \leq y < M$.

Last, input the string "Route T mode", where T is the expected running time. The program will terminate at the first iteration after T seconds (or when it cannot find better answers), and "mode" can be "Fast", "Balanced" or "Precise". The difference of the three strategies will be introduced later. If not explicitly specified, the mode will be automatically set "Balanced".

Input example: (exactly the example plotted in individual project document)

```
8 8
Obstacle 1 3 Obstacle 1 4
```

```
Obstacle 3 1 Obstacle 3 2 Obstacle 4 1 Obstacle 4 2
Obstacle 6 3 Obstacle 6 4 Obstacle 7 3 Obstacle 7 4
Terminal 3 0 2 5 6 7 6
Terminal 3 1 1 1 6 5 4
Terminal 3 2 0 4 3 7 1
Route 60 Fast
```

1.3 Output Format

The program will output the routed board ($N \times M$ elements). -1 represents obstacles, 0 represents unused grids, positive integers represent used grids, and different integers are for different terminal sets.

Output example: (exactly the output of the input example)

```
0 0 1 1 1 1 1 1
0 2 0 -1 -1 2 2 1
3 2 2 2 2 2 0 1
3 -1 -1 0 2 0 0 1
3 -1 -1 3 2 0 0 1
3 3 3 3 2 0 1 1
0 3 0 -1 -1 0 1 0
0 3 0 -1 -1 0 1 0
```

The program will also plot a visualized result on console.

2 Algorithm Introduction

2.1 Integer Programming Formulation

Let T_i denote the i-th terminal set, $\mathcal{T}(T_i)$ denote all Steiner trees connecting set T_i (even including those crossing obstacles).

For each tree of each terminal set $t \in \mathcal{T}(T_i)$, we create a variable x_{it} denoting whether tree t is used, and a constant c_{it} denoting the number of used grids on tree t. For each terminal set T_i, we create a variable s_i denoting whether set s_i is NOT routed.

For each grid v, let constant u_v denote the capacity of grid v, i.e. 1 for normal grids and 0 for obstacles, and for each tree t, let constant a_{vt} denote whether tree t contains grid v.

As a result, x_{it} and s_i are all integer (even binary) variables. Let M be a sufficiently large number, then our goal is:

$$\min_{x,s} \sum_{i=0}^{P-1} \sum_{t \in \mathcal{T}(T_i)} c_{it} x_{it} + M \sum_{i=0}^{P-1} s_i$$

such that

$$\forall i = 0, \cdots, P-1, s_i + \sum_{t \in \mathcal{T}(T_i)} x_{it} = 1$$

$$\forall v \in \{0, \cdots, N-1\} \times \{0, \cdots, M-1\}, \sum_{i=0}^{P-1} \sum_{t \in \mathcal{T}(T_i)} a_{tv} x_{it} \leq u_v$$

Since all the constraints are linear in respect to the variables, this is a integer programming problem.

2.2 Linear Programming Approximation and Column Generation

2.2.1 Column Generation

It is too expensive to find the whole sets $\mathcal{T}(T_i)$, but if we only consider their subsets $\mathcal{S}(T_i) \subset \mathcal{T}(T_i)$ instead of $\mathcal{T}(T_i)$ itself, we can get approximate solutions.

The sets $\mathcal{S}(T_i)$ are initially all empty, or containing some primal trees created by some generator. The initial sets depend on the strategy specified ("Fast",

"Balanced" or "Precise"). The details will be introduced later.

For each iteration process, the linear programming (rather than integer programming) is solved by Gurobi, which yield (primal) solutions $(\hat{x}, \hat{s})$ and also dual solutions $(\hat{\lambda}, \hat{\pi})$. For each grid v, $\hat{\pi}_v$ describes how much the target function will vary per more capacity of v given. For each terminal set i, $\hat{\lambda}_i$ means that only a new tree t of terminal set T_i with its $\sum_v a_{tv}(1 - \hat{\pi}_v) < \hat{\lambda}_i$ will be possible to improve the target function. Note that $\hat{\pi}_v \leq 0, \forall v$, since that the target function is sure to decrease or hold when increasing u_v.

Based on that, we have three methods to generate new columns. The first one is based on existing trees and the dual values, and the others are based on the solution using current trees and directly find some possible trees "out of nothing" (rather than out of existing trees). When all three methods cannot generate new trees, the algorithm calls the integer programming solver and terminates.

2.2.2 Generating with Existing Trees and Dual Values

We consider different sets of terminals T_i separatedly. The goal is to generate a new tree t with $\sum_v a_{tv}(1 - \hat{\pi}_v)$ as small as possible. If the smallest value is still greater or equal to $\hat{\lambda}_i$, then no tree is generated here.

We start by enumerating all known trees of current terminal set $\mathcal{S}(T_i)$, and then try to remove some grids of the tree and replace with new ones. We still enumerate which grid we remove.

After removing this grid, we should also remove those "useless" grids (the degree of the grid is 1 and is not a terminal). The tree is now broken into several pieces. Different from the original paper, the number of pieces can be more than 2, which means we should find an optimal Steiner tree rather than a path.

We have an approximate approach here. Since the tree is broken only up to 4 pieces (the degree of any grid is at most 4), let's assume only one Steiner point is needed. Define grid v's weight is $(1 - \hat{\pi}_v)$, we perform Dijkstra's shortest path algorithm from each broken piece in respect of weight $(1 - \hat{\pi}_v)$, and enumerate the Steiner point, then we can find an approximate solution to the best Steiner tree.

If the total weight of this connected tree is less than $\hat{\lambda}_i$, we add this tree to the tree set of current terminal set $\mathcal{S}(T_i)$.

As an optimization, if we choose to remove a certain 2-degree grid, it is equivalent to remove any grid within its 2-degree adjdacent branch (which seem to be on the same "edge"). Thus, we enumerate grids in an alternative way, recording which grids we have removed, in order to avoid recomputation.

Since this generating method need many enumerations (all sets, all trees, all removing grids, all possible Steiner points), the method appears relatively slow. And sometimes this method cannot generate some "obvious" trees. If we use only this method, the solution will be so terrible that it even cannot route a path of some terminal set which will not affect other sets! As a result, this method is only performed when the other two methods cannot generate new trees.

2.2.3 Generating with Adjusting Current Solution

This method is based on the integer programming solution, rather than (relaxed) linear programming solution (or its dual).

We only consider the terminal sets which are successfully routed in current solution. Say we are considering terminal set T_i routed with tree t.

We remove tree t from the solution, and respect all other routed trees as obstacles. We find another Steiner tree connecting T_i, not crossing obstacles, and overlapping tree t as few as possible, then as short as possible. Weighting grid v with 1 for empty grids, M for grids in t, M^2 for obstacles, the tree is found approximately by Dijkstra's shortest path algorithm like the fashion above.

Regardless of whether the tree satisfy the limitation $\sum_v a_{tv}(1 - \hat{\pi}_v) < \hat{\lambda}_i$, as long as the tree is not in current set $\mathcal{S}(T_i)$, we will add it there.

This method is pretty fast, since it only calculates one integer programming (tens of milliseconds in practice) and only performs several Dijkstra's algorithms, of high efficiency.

2.2.4 Generating with Satisfying Other Trees

This method is also based on the integer programming solution. If possible, we would like a tree of a certain terminal set to sacrifice the cost (or length) itself, to satisfy other trees.

We still start with each terminal set T_i. We temporarily clear $\mathcal{S}(T_i)$, and perform run integer programming solver for the solution, so the (currently) best solution without trying to route T_i will be given. We now find a Steiner tree connecting T_i, not crossing obstacles, and overlapping current solution (without T_i) as few as possible, then as short as possible. Similarly, weighting grid v with 1 for empty grids, M for grids in the solution above, M^2 for obstacles, the tree is found approximately by Dijkstra's shortest path algorithm like the fashion above.

Since the method needs to calculate several (P) instances of integer programming, it is not so fast than the previous one. However, it can still finish in less than 1 second per iteration, which is fast enough.

2.3 Why I Did Not Use FLUTE

The FLUTE is used to calculate the shortest Steiner tree of a certain terminal set. However, it did not take the obstacles or other terminal sets into account. This can lead to worse time performance and/or even worse solution quality, because the generated tree is so "selfish" that, as long as the tree is considered in the set, the linear programming will try to accept its request (only to route itself). Moreover, if we add too many "not so useful" trees to the set, the quality of the final solution will decrease, due to the attributes of the "approximate linear (rather than integer) programming formulation".

Anyway, as a result, FLUTE is not used since it is not helpful to the global solution quality. If we were to calculate only one set of terminal with a lot of points, only a few obstacles are given, we would want to try FLUTE. For at least two (randomly generated) sets of terminals, FLUTE is not helpful any more.

2.4 Calculation Strategies

The differences of the three strategies are the difference between initial tree sets, and the different strategies for column generation.

In detail, the set for "Fast" or "Balanced" mode is empty, while for "Precise" mode also contains other trees generated from every grid of the board.

"Fast" mode is suggested when routing large (more than 20×20) complex boards, or when you do not want to wait long (more than a minute). It can quickly give a decent solution where one cannot find any "obvious" improvements, but may sometimes be stuck at a local (rather than global) optimal solution and cannot get itself out. This mode is REALLY FAST, which can finish in seconds for not-so-large (less than 20×20) boards.

"Balanced" mode is suggested when routing relatively small (less than 10×10) or simple boards, or when you plan to wait for a some time (from minutes to 1 hour). The solution will improve slower but the final result can be better. It can also be tried when "Fast" mode runs very fast and did not give a satisfying answer.

"Precise" mode is only suggested when routing very small (less than 8×8) boards, or when you expect to get very good solution and can wait for hours for larger boards. The solution will improve very slow, and need hundreds of minutes to show its advantage.

For any strategy, if you do not want to wait any more, you may cancel it at any time, and the program will output the current best solution.

3 Object-Oriented Programming Design

3.1 Classes

I designed classes of several different modules:

- Utility classes.

  This module includes a template `class Matrix<T>` (inheriting

`Vector<Vector<T>>`), and also some non-member functions such as `OStream &operator <<(Ostream &, const Vector<T> &);`.

- Basic classes.

  These classes are `class Board` (representing a complete board with obstacles and terminal sets information), and `class Tree` (representing a single Steiner tree).

  There are also provided some debug functions such as `OStream &operator <<(OStream &, const Board &);`, and `bool Tree::valid();`.

- Interface classes.

  These classes are `class TreeGenOne`, `class TreeGenAll` (providing interface to generating one or all Steiner trees of a certain terminal set), and `class Router` (providing interface to solving the original problem).

  The `class TreeGenOne` and `class TreeGenAll` have default implementations (rather than declaring pure virtual functions) with brute-force methods. `class TreeGenAll` is not yet inherited since it is not used by column generation algorithm, but I still left this interface for better code reusing.

- Implementation classes (of interfaces).

  This include the needed classes of the algorithm such as `class ColumnGen` (implementing `class Router`), and also some classes helpful to validate solutions, `class StupidIP`, to search all Steiner trees by brute-force then call the integer programming solver and get the answer.

3.2 Files Organization

- Directory `/src`

  This directory contains the source code of the solver. It also contains `makefile`, and `gurobi65.dll` used by Gurobi optimizer. The generated `main.exe` are also placed here.

  - Directory `/src/include`

Store headers. `gurobi_c++.h` and `gurobi_c.h` are for Gurobi, `global.h` for including external headers, and others are declarations of classes.

- Directory `/src/lib`

  Store library files of Gurobi: `gurobi_c++mt2012.lib` and `gurobi65.lib`.

  Other files in directory `/src` are corresponding `.cpp` files implementing each class, and also `main.cpp` defining main function.

- Directory `/testcase`

  There are inputs and outputs for test cases.

  Test generator is also placed here. The reason will be explained later.

- Directory `/doc`

  Storing this document and its TEX source.

As a result, the files are arranged neat and clean. :)

3.3 Mentionable Design Principles

- All destructors of classes with any virtual functions are virtual.
- No `using namespace std;`, even no `using std::some_type;`. All used STL types are in aliases such as `Vector`, `OStream`.
- The classes which can have extensions have some proper virtual functions, though at present no class inherits them.
- There are enough comments that anyone who understands the algorithm will understand the code with ease.
- `git` is used for version control.
- The classes itself verify the correctness of the inputs (parameters) and results. The `main.cpp` also checks the input and give friendly information. In addition, `gen.cpp` is placed in directory `testcase`. It is not considered to be

in the main project because it is not expected to call the project's functions. This can avoid generator going wrong just because the main project's function is wrong. This error can sometimes invisible because two samely-wrong functions can think each other is right!

3.4 Core Functions

Since the comments in the code are pretty in detail, and the algorithm is fully introduced above, the implementation details are not introduced here. Here is only the definitions of the core functions.

The core functions of `class ColumnGen` are:

```
// columngen.h
#ifndef _COLUMNGEN_H
#define _COLUMNGEN_H

#include "router.h"

class ColumnGen: public Router{
public:
    String m_mode;
    ColumnGen(const Board &board, String mode):
    Router(board), m_mode(mode){
    }
    virtual ~ColumnGen(){
    }
    /*
        The core routing function.
    */
    virtual Vector<Tree *> route(int) const;
protected:
    struct expandFinished{
    };
```

```
22      /*
23          Call Gurobi and solve the linear
24          programming problem. mode = 0 for
25          [0, 1], mode = 1 for {0, 1}.
26      */
27      double solveLP(
28          Vector<Vector<Tree>> &treesets,
29          int mode,
30          Matrix<double> &mapPi,
31          Vector<double> &vecLambda,
32          int ignoreIdx = -1
33      ) const;
34      /*
35          Given a tree and the Pi matrix,
36          calculate the total weight of the tree.
37      */
38      static double CalcTreeLambda(
39          const Matrix<double> &mapPi,
40          const Tree &tree, int n, int m
41      );
42      /*
43          Try to generate at most "r" new trees of
44          terminal set "idx" to "treeset".
45      */
46      bool expand(
47          const Matrix<double> &mapPi,
48          double lambda,
49          Vector<Tree> &treeset,
50          int n, int m, int idx, int &r
51      ) const;
52      /*
53          Try to generate at most "r" new trees of
54          terminal set "idx" from "tree" to
```

```
55          "treeset".
56      */
57      bool expand(
58          const Matrix<double> &mapPi,
59          double lambda,
60          Vector<Tree> &treeset, const Tree &tree,
61          int n, int m, int idx, int &r
62      ) const;
63      /*
64          Remove all non-cut grids of "tree",
65          except terminals "idx" of "map",
66          and except the grid ("exx", "exy").
67      */
68      static void RemoveNonCuts(
69          const Matrix<int> &map, int idx,
70          Matrix<bool> &tree, int n, int m,
71          int exx = -1, int exy = -1
72      );
73      /*
74          Remove all non-cut grids of "tree",
75          except terminals "idx" of "map",
76          mark all removed grids in "visited",
77          and return the total removed weight
78          in regards of "mapW".
79      */
80      static double RemoveNonCuts(
81          Matrix<bool> &visited,
82          const Matrix<int> &map, int idx,
83          const Matrix<double> &mapW,
84          Matrix<bool> &tree, int n, int m
85      );
86      /*
87          Try to generate at most "r" new trees of
```

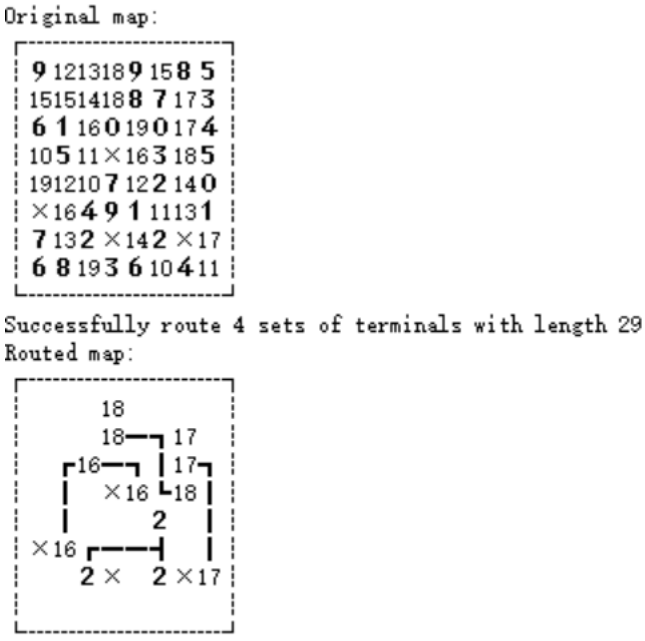
```
88          terminal set "idx" from "base" (the
89          broken pieces of a tree) to "treeset".
90      */
91      bool expand(
92          const Matrix<double> &mapW,
93          double lambda,
94          Vector<Tree> &treeset,
95          const Matrix<bool> &base,
96          int n, int m, int idx, int &r
97      ) const;
98      /*
99          Perform Dijkstra's algorithm from source
100         "base" and in respect of weight "mapW".
101     */
102     Matrix<Pair<double, Matrix<bool>>> dijkstra(
103         const Matrix<bool> &base,
104         const Matrix<double> &mapW,
105         int n, int m, int idx
106     ) const;
107     /*
108         Try to generate and insert the best tree
109         of terminal set "termset", "idx" in
110         regards of "mapW".
111     */
112     bool suggestTree(
113         const Vector<Vector<Point>> &termsets,
114         Vector<Vector<Tree>> &treesets,
115         const Matrix<double> &mapW,
116         int n, int m, int idx
117     ) const;
118     /*
119         Generate the best tree of terminal set
120         "termset", "idx" in regards of "mapW".
```

```
121     */
122     Tree suggestTree(
123         const Vector<Point> &termset,
124         const Matrix<double> &mapW,
125         int n, int m, int idx
126     ) const;
127 };
128
129 #endif // _COLUMNGEN_H
```

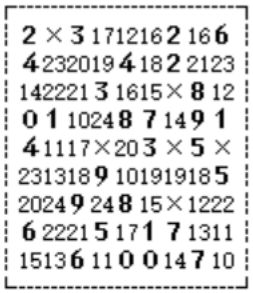
4 Results

These test cases are randomly generated, and is routed with "Fast" mode.

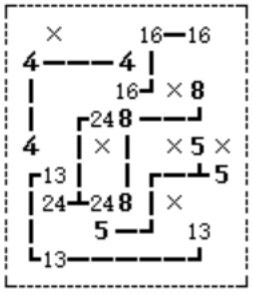
The (visualized) results in brief are below. See `testcase` directory for details.



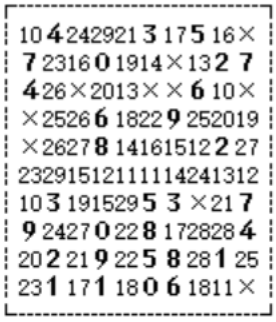
Original map:



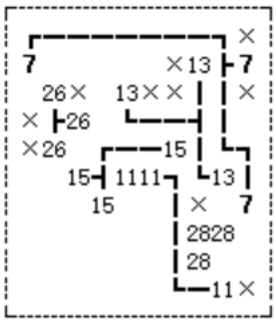
Successfully route 6 sets of terminals with length 51
Routed map:



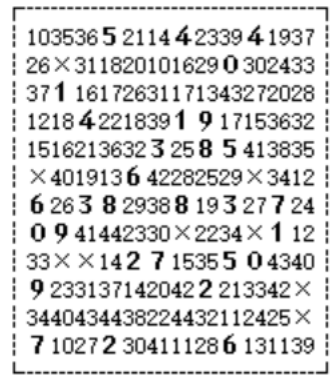
Original map:



Successfully route 6 sets of terminals with length 51
Routed map:

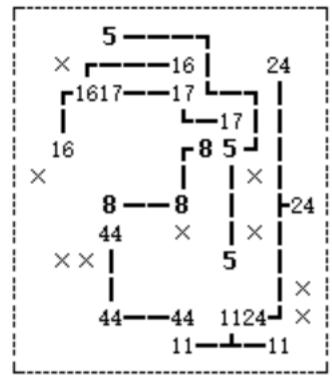


Original map:

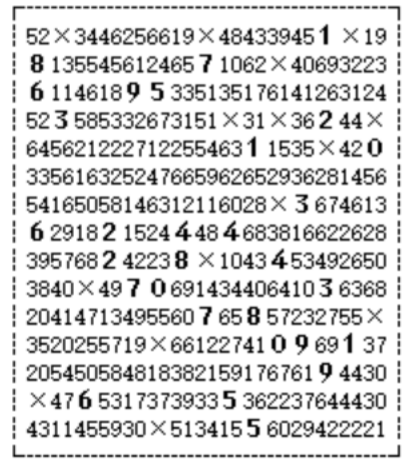


Successfully route 7 sets of terminals with length 64

Routed map:

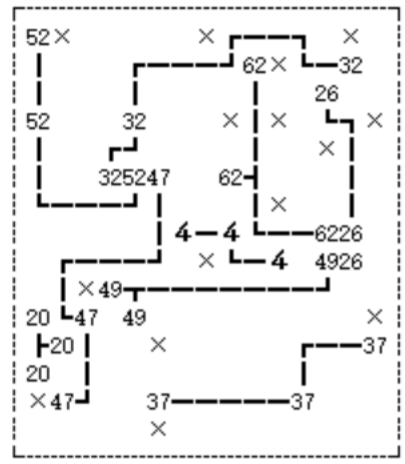


Original map:

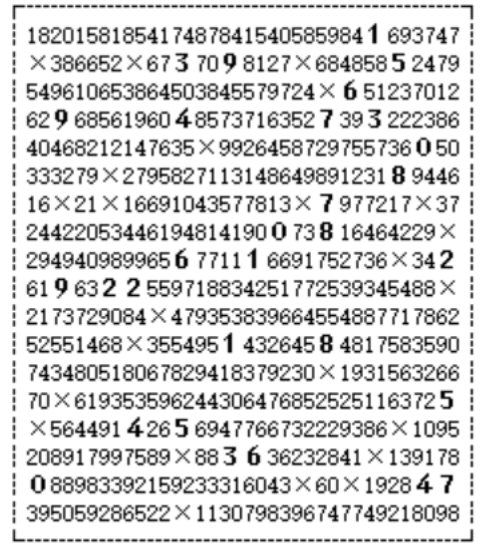


Successfully route 9 sets of terminals with length 97

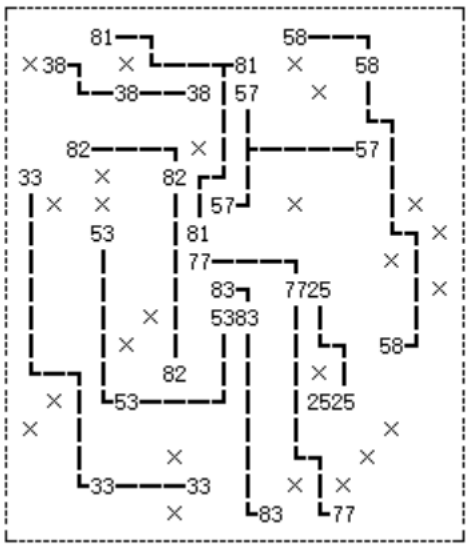
Routed map:



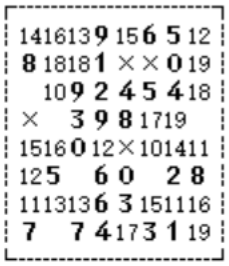
Original map:



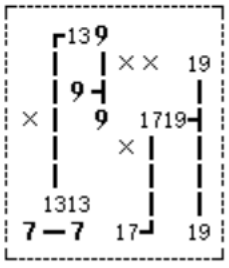
Successfully route 10 sets of terminals with length 133
Routed map:



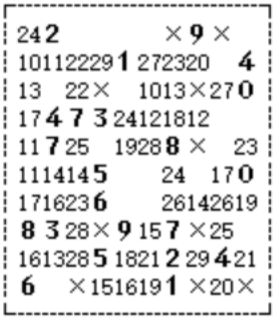
Original map:



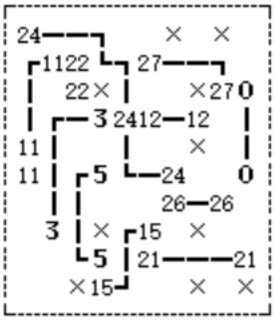
Successfully route 5 sets of terminals with length 31
Routed map:



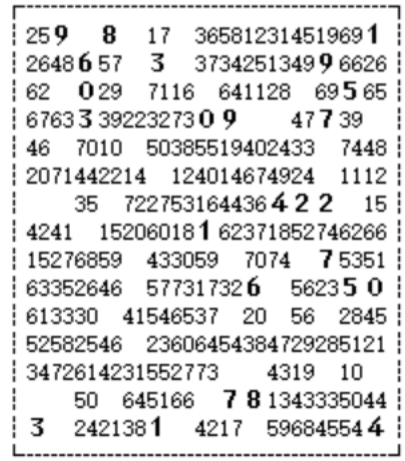
Original map:



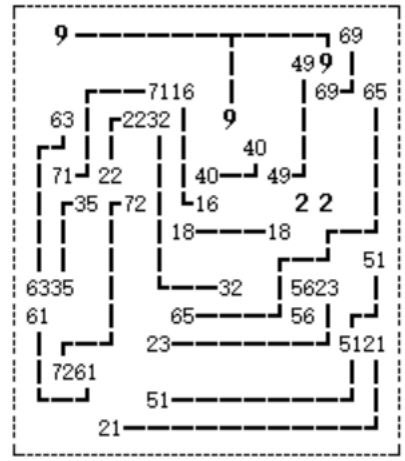
Successfully route 11 sets of terminals with length 58
Routed map:



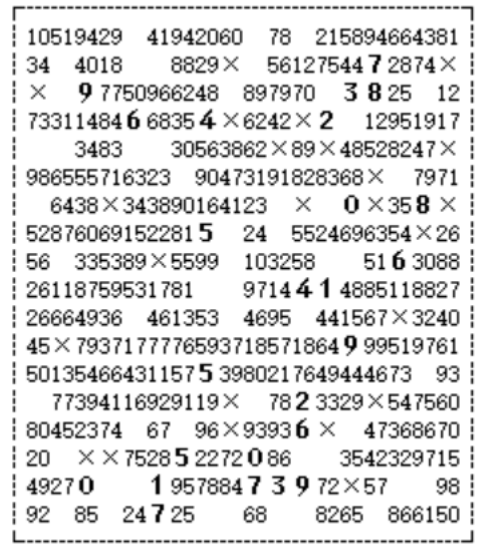
Original map:



Successfully route 19 sets of terminals with length 154
Routed map:



Original map:



Successfully route 19 sets of terminals with length 155
Routed map: