Escape Routing – Network Flow and Heuristics

Hong Wenhao, 2016011266 Sun Zhenbo 2016011277 Hu Zhiyuan 2016011260

June 22, 2017

Contents

0 Build, Run and Test

After unzipping the code, there are three directories.

doc The documentation is stored in this directory.

src The source files are stored in this directory.

testcase The test results, including a built visualizer (written in Java Swing), are stroed in this directory.

0.1 Building and Running

The project uses CMake to manage the building process. To build the project, run the following scripts in */src*.

```
mkdir build
cd build
cmake ..
make
```

The executable will be named "src" in the directory */build*.

To run, use the following command in */build*.

```
./src
```

0.2 Testing

The default entrance function ("main()") accepts the dimensions of the board and saved output in */testcase/*. However, users can customize behavior by modifying *Common/main.cpp*.

Solver The solver class implements the solving process and produce verbal output. Along with Routers it implements a strategy design pattern.

Timer The timer is a handy proxy for solver, which not only run the solution, but also save the output in *../testcase/* directory.

Visualizer The visualizer can be runned by using

```
java -jar visualizer.jar
```

It is recommended to use it with terminals, because error messages will be shown in the terminal(if errors occur).

Validator A validator is written (stored in */testcase*) to validate output. It checks whether each internal node is successfully routed without crossing, and checks if there are "scattered" edges on the board. It also checks whether the outputs of the three routers are consistent.

0.3 Screenshots

0.3.1 Entrance function

```
[matt154@Hongde-MacBook-Pro:src] $ mkdir build
[matt154@Hongde-MacBook-Pro:src] $ cd build
[matt154@Hongde-MacBook-Pro:build] $ cmake ..
-- The C compiler identification is AppleClang 8.0.0.8000042
-- The CXX compiler identification is AppleClang 8.0.0.8000042
-- Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc
-- Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++
-- Check for working CXX compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/matt154/Documents/OOP/EscapeRouters/src/build
```

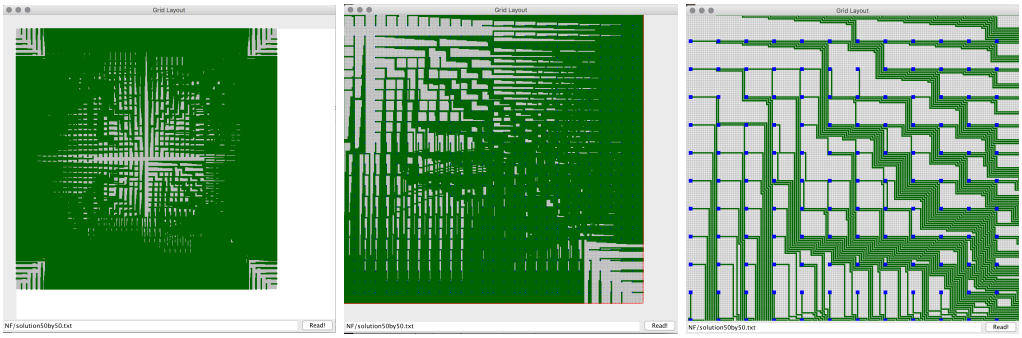
Figure 1: Building (CMake)

```
[matt154@Hongde-MacBook-Pro:build] $ make
Scanning dependencies of target src
[ 10%] Building CXX object CMakeFiles/src.dir/Common/Board.cpp.o
[ 20%] Building CXX object CMakeFiles/src.dir/Common/Router.cpp.o
[ 30%] Building CXX object CMakeFiles/src.dir/NetworkFlow/NFRouter.cpp.o
[ 40%] Building CXX object CMakeFiles/src.dir/Common/main.cpp.o
[ 50%] Building CXX object CMakeFiles/src.dir/Common/Solver.cpp.o
[ 60%] Building CXX object CMakeFiles/src.dir/DivideConquer/Quarter.cpp.o
[ 70%] Building CXX object CMakeFiles/src.dir/DivideConquer/DCRouter.cpp.o
[ 80%] Building CXX object CMakeFiles/src.dir/Common/Timer.cpp.o
[ 90%] Building CXX object CMakeFiles/src.dir/Rule/RuleRouter.cpp.o
[100%] Linking CXX executable src
[100%] Built target src
[matt154@Hongde-MacBook-Pro:build] $ ./src
Please enter dimension (m x n) ...
Default option is to use Rule Router for m = n, and NetworkFlow Router for m != n
... 10 10
Timing on 10 by 10 type is RU

Start solving ...
2 cannot do
shortest is 3
Successfully routed 100
Cost is 976

Used 0.00061 secs to route.
```

Figure 2: Making & testing

0.3.2 Visualizer

The visualizer can display the routed board with different scales.



(a) 1x (b) 2x (c) 4x

Figure 3: Visualizer, with NF router results

The usage is simple, to zoom in near a specific point, left click. To return to original scale, right click. Notice that the image is zoomed in with the position clicked at the upper left corner.

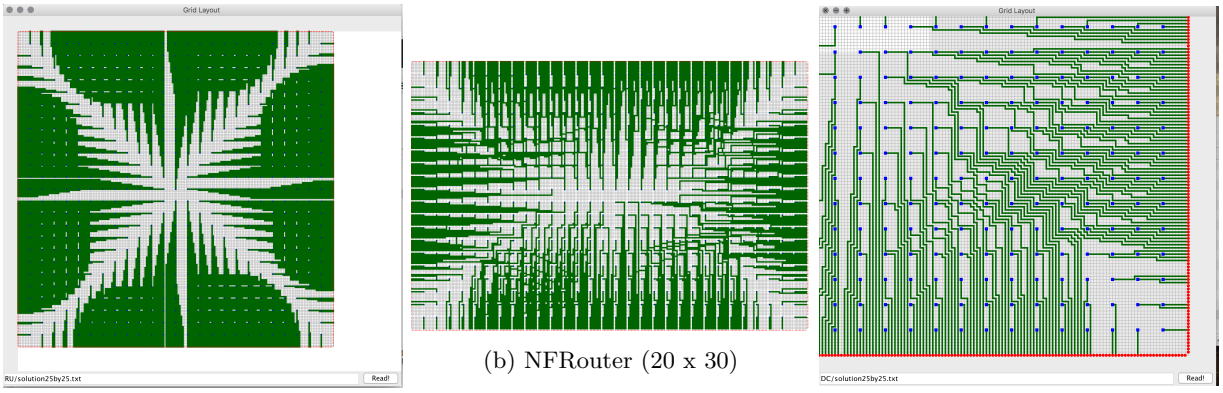
1 Overview

In this project, an escape routing problem is solved using three approaches, each with different performance and limitation. The escape routing problem is given as follows.

Given equally spaced internal node array

a find the smallest number k of "gap" lines between the nodes, such that the nodes can connect to terminal nodes with non-crossing paths.

b Furthermore, find the shortest routing paths if possible.



(a) RuleRouter (20 x 20)

(b) NFRouter (20 x 30)

(c) Partial solution

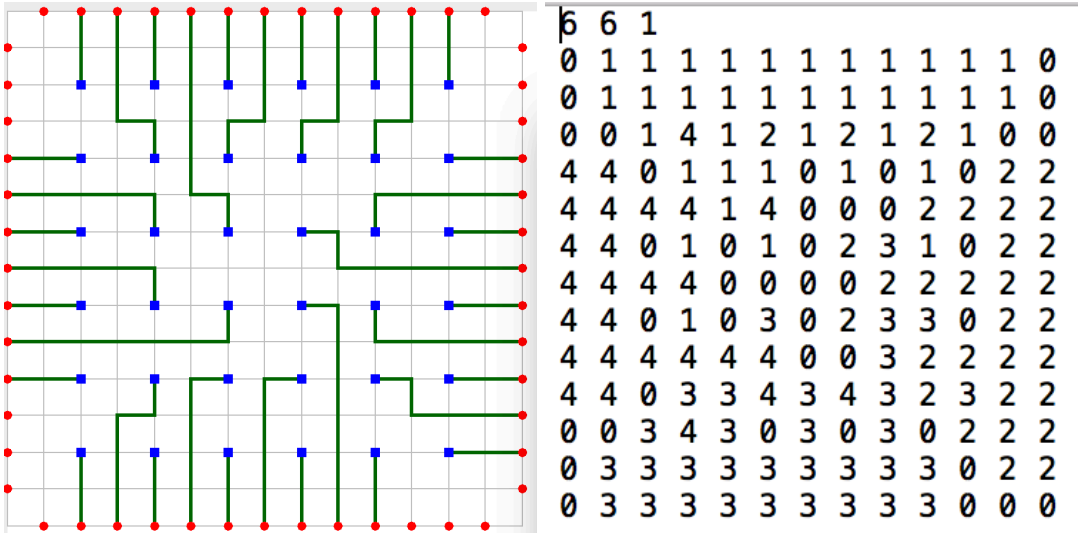
Figure 4: Different routing solutions

In this project, the number of nodes is assumed to be about 400 (20×20) to ~5000 (70×70). However, other number of nodes (e.g. 100×100) can also be handled by the solvers, see the test data for details.

2 Data Representation

2.1 Board format

Each board is encapsulated by a Board object, which keeps track of the size of the problem (N, M, K, the height, width and gap respectively) and the real dimesion of the board (DM, DN). In each grid of the board, a number from $\{0, 1, 2, 3, 4\}$ is kept, which tells the direction of the edge from that grid. An example is illustrated below.



```
6 6 1
0 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 0
0 0 1 4 1 2 1 2 1 2 1 0 0
4 4 0 1 1 1 0 1 0 1 0 2 2
4 4 4 4 1 4 0 0 0 2 2 2 2
4 4 0 1 0 1 0 2 3 1 0 2 2
4 4 4 4 0 0 0 0 2 2 2 2 2
4 4 0 1 0 3 0 2 3 3 0 2 2
4 4 4 4 4 4 0 0 3 2 2 2 2
4 4 0 3 3 4 3 4 3 2 3 2 2
0 0 3 4 3 0 3 0 3 0 2 2 2
0 3 3 3 3 3 3 3 3 3 0 2 2
0 3 3 3 3 3 3 3 3 3 0 0 0
```

(a) Board

(b) Digital Board

Figure 5: Board representation

2.2 Internal Object Design

The whole project is divided into four modules, the most important of which is the Common module. More implementation details and algorithms are explained in Section 3.

2.2.1 Common

In the common module, several basic objects are stored.

Board Board class is already discussed in Section 2.1

Router Router class is a template class for the routing process, it has two virtual methods. Function OK() returns whether the board is routable, and function route() returns a pointer to a solution board, or NULL if it does not exist.

Solver The solver, along with router class, implements a strategy pattern. It uses incremental search for K (starting from $\lfloor \frac{n}{4} \rfloor - 1$) instead of binary search, because by experiment k is commonly 2 or 3 apart.

Timer The timer uses the solver class and can be seen as a proxy. Its purpose is to keep log of the results and do output.

Java/Drawer The drawer is written in Java Swing and do visualization, explained in Section 0.3.2.

2.2.2 NetworkFlow

In the NetworkFlow module is stored the implementation of NFRouter. It utilizes network flow formulation.

2.2.3 DivideConquer

In the DivideConquer module is stored the DCRouter, which uses divide and conquer technique to speed up solution. Another class Quarter, which is a modification of NFRouter by inheritance, is stored in the module and used by DCRouter.

2.2.4 Rule

The RuleRouter, which is the fastest router, is stored in this module.

3 Routing approaches

The routing approahes are explained briefly in this section. For more implementation details please consult */src*. The performances and correctness are dicussed in Section 4.1.

It is easy to prove that minimum gap is at least $\lfloor \frac{n}{4} \rfloor - 2$ for square boards.

Proof. There are n^2 internal nodes, and $kn + n + k$ terminals on each edge. Therefore

$$\begin{aligned} 4kn + 4n + 4k &\geq n^2 \\ k &\geq \frac{n^2 - 4n}{4n + 4} \\ k &\geq \frac{n^2 + n}{4n + 4} - \frac{5n}{4n + 4} \geq \frac{n}{4} - 2 \end{aligned}$$

□

By experiment this gap is not too far away from $\lfloor \frac{n}{4} \rfloor - 2$, so using incremental search can be more efficient than binary search.

3.1 Network Flow

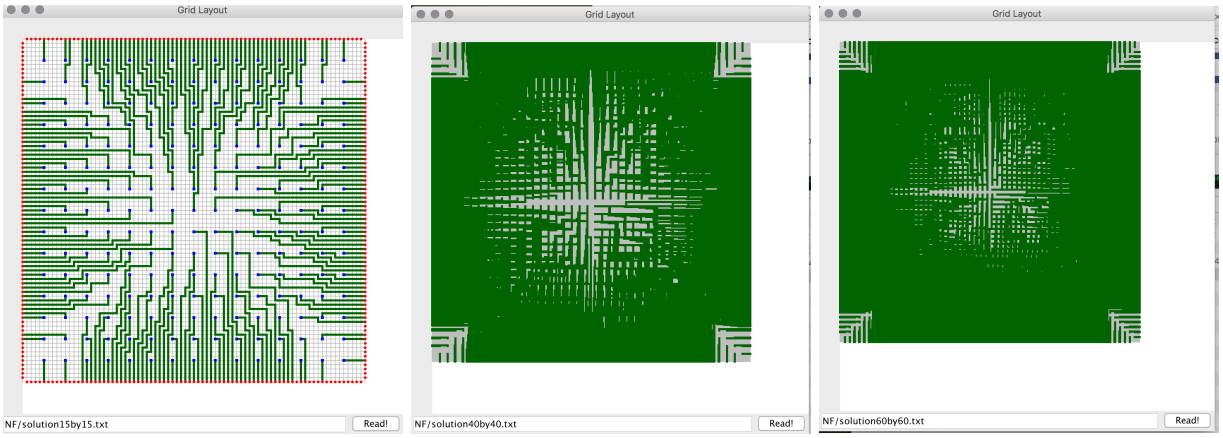
The formulation to network flow is straightforward. The first approach is to treat every intersection point of the board as a node on the network flow graph, and add a "super source" connected to every internal nodes, a "super sink" connected to every node on the boundary. Every edge between the nodes have capacity one, and our goal is to make the flow $M \times N$.

However, the above formulation only forbids the same edge to be used twice, which does not actually forbid "crossing paths" on the board. Also, the formulation cannot deal with the total length. So another formulation using minimum cost flow is invented.

The correct approach is to assign the capacity 1 to both the nodes and the edges, and a cost of 1 to the nodes. Nodes with capacities and costs are implemented in a classical way – by splitting a node into an "in" node and an "out" node, and connecting them by an edge with capacity 1 and cost 1. Note that the length of the escaping path is correctly recorded by the number of nodes it passed through.

Having a correct formulation, we implemented the augmenting shortest path algorithm, which uses label-correcting algorithm, as dicussed on this site [1].

Several outputs are the following. Note that this router can not only handles squares, but also rectangles.



(a) 15 x 15 (b) 40 x 40 (c) 60 x 60

Figure 6: NFRouter with different sizes

3.2 Divide and Conquer

After studying the solution from NFRouters, we found that optimal solutions for squares have some special properties. One such properties is that the four corners are sparse and the other regions are extremely dense. However, this property might not be so useful for speeding up the solution.

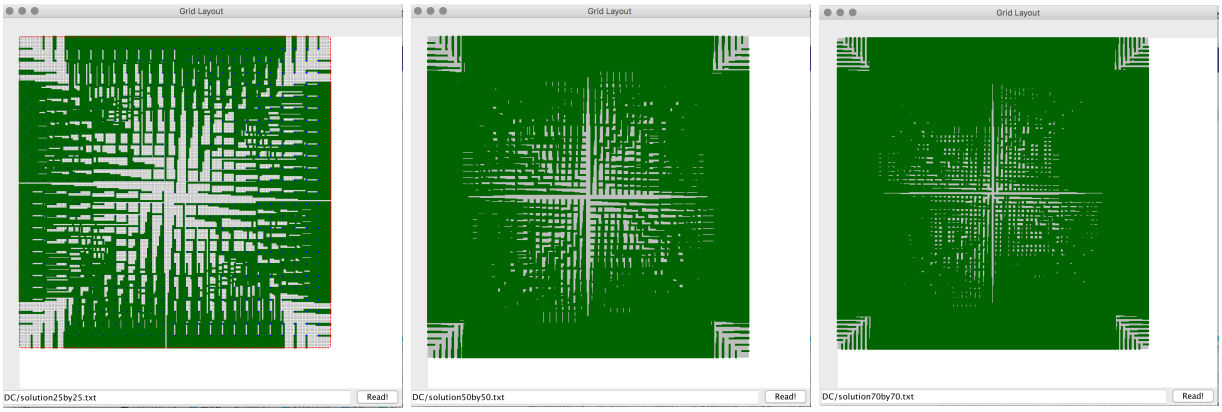
Squares are special in that the are more regular and symmetric compared to rectangles, and we found that the solutions also have some sort of symmetry for the four corners. It seems that we can divide the whole board into four regions, solve for one quarter, and rotate the solution to get the whole results. This leads us to the DCRouter.

To implement such an approach and do better code reuse, we have class Quarter inherit from NFRouter. We can use the same routing function, and only need to modify the add_edge() and fulfill() methods, which are designed for modification, to make the underlying graph different.

Detail implementations includes even-odd case analysis and are not covered in this documentation. It is worth noting that after Quarter behaves correctly, what is left is to do rotation.

Theoretically this approach can speed up the efficiency of network flow by a constant factor ($\sim 4^2 = 16$). In reality DCRouter is about 20 times faster.

Another thing to be noted is that, when n is odd, DC router returns suboptimal length (k is optimal) because the center point cannot be easily divided into any quarter. For even size, the router behaves optimal as verified by NFRouter.



(a) 25 x 25 (b) 50 x 50 (c) 70 x 70

Figure 7: DCRouter with different sizes

3.3 Rule Based Method

3.3.1 Observations

The previous two methods have a worst case complexity of $O(fVE) = O(n \times n^8) = O(n^9)$, but normally runs in $\sim O(n^5)$. However, these complexities are not affordable as n grows large. So we proposed a rule-based routing methods, which utilizes the symmetry of squares.

We have mentioned that, it seems that the paths can be safely divided into four quarters. This suggestion is verified by DCRouter results in3.2. Nevertheless, points at the boundary of the quarters may not be able to be properly distributed. This leads us to think of another way of dividing the board – by the diagonal. Let's take a closer look at the NF solution.

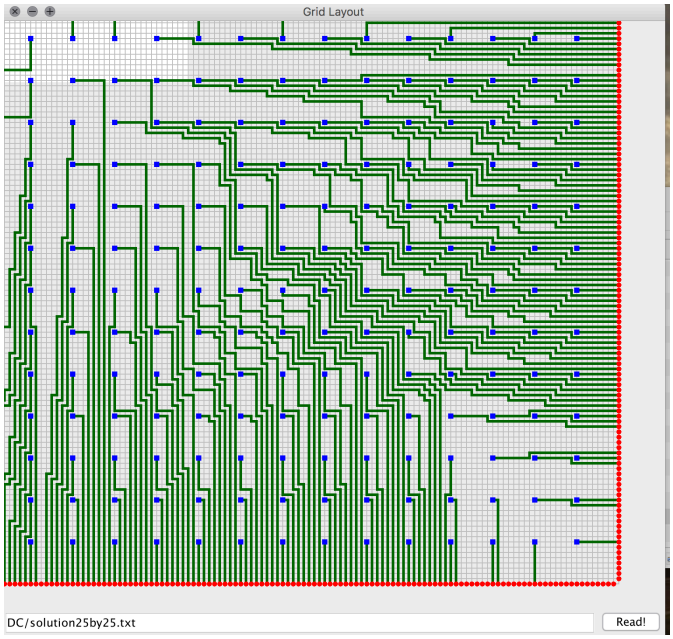


Figure 8: Solution at the lower right quarter

We can see a bunch of paths stacking at the diagonal, does it mean that we cannot divide the solution along the diagonal?

No. The key thing to notice is that, the nodes under the diagonal is routed to the bottom, and the nodes above the diagonal is routed to the right.

For the lower part, we can "pull" the paths down to the bottom without changing its length. If a shortest solution exists, there should be one solution, in which the paths are preferrably taking down edges. By doing so

we should be able to clear the diagonal, and to judge that no solution exists if the paths have to cross over the diagonal.

Another important observation is that, the nodes at the center axis are spread evenly right and left (this is becuase the axis has $\frac{n}{2}$ nodes, and the gap is about $\frac{n}{4}$), and the nodes at the top of this "quarter" mountain are extending their paths to the right, occupying other columns' terminals. This is just a heuristics, but turned out to perform quite well.

3.3.2 Rules

The routing process is done in the following manner. First, we keep a 2D boolean array recording whether a node is visited (similar to BFS).

i Distribute the terminals near at the bottom center to the nodes that are near the center axis.

ii Search paths one by one for the right half. Prefer down edges to right edges when there are options. Note that sometimes, we need to backtrack, this is not a matter because we never visit a grid twice.

iii Then do the same for the left half.

iv rotate the quarter 3 times to fill the complete board.

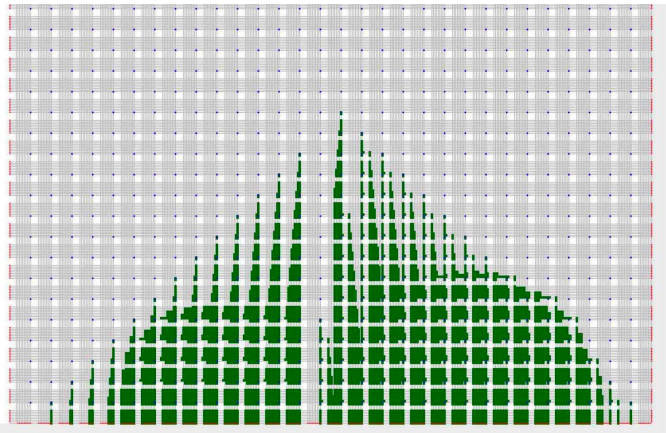


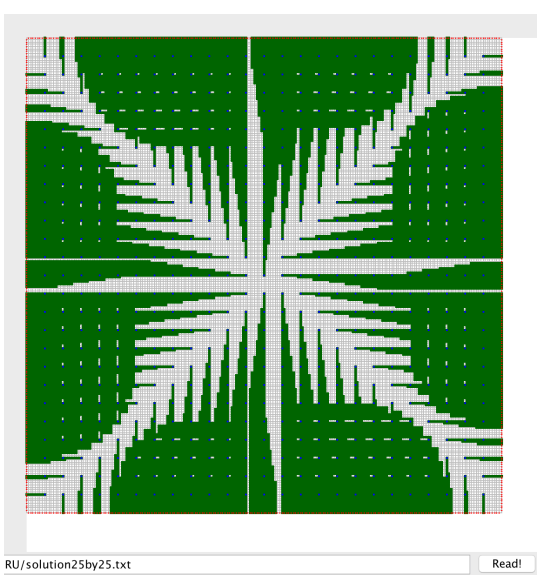
Figure 9: lower quarter (center axis not shown)

The searching order in ii is not arbitrary. For example, if we route the upper points first, we would have blocked the lower points from escaping. Inspired by Figure 8, we devised the following order. (For the right half, same rule for the left half)

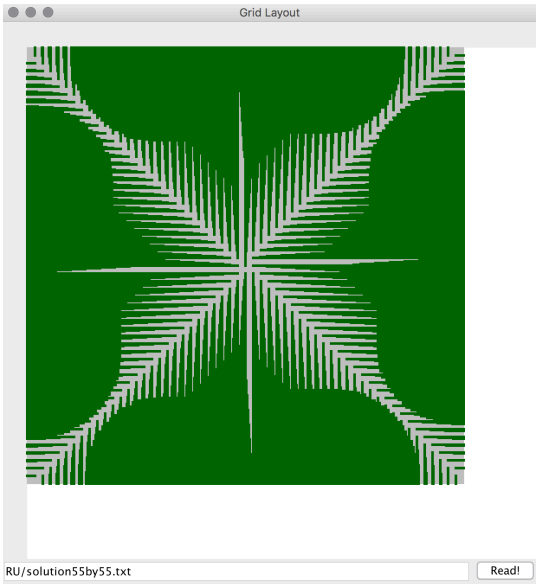
1 Route the points column by column, from left to right.

2 For each column, route the points from bottom to top.

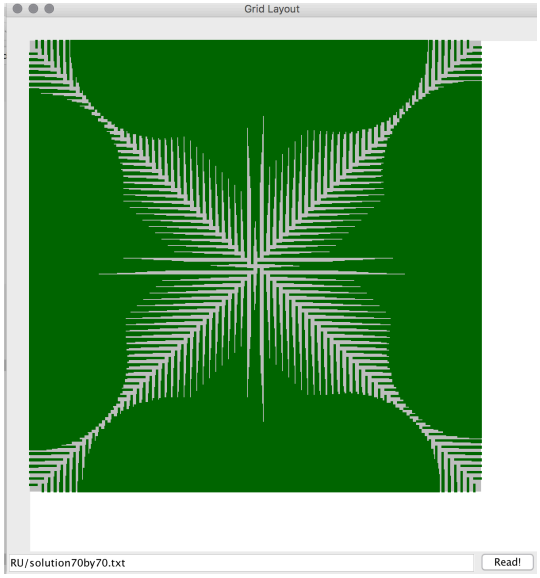
As there are not much terminals in each column, we need to "push" some paths to the right. Whenever we enter another "tunnel" of gap, we need to ensure that the nodes lower in that tunnel are all routed, otherwise we may block these nodes from escaping. This is done by making delicate recursive calls and will not be further illustrated in this text, please consult */src/Rule* for details.



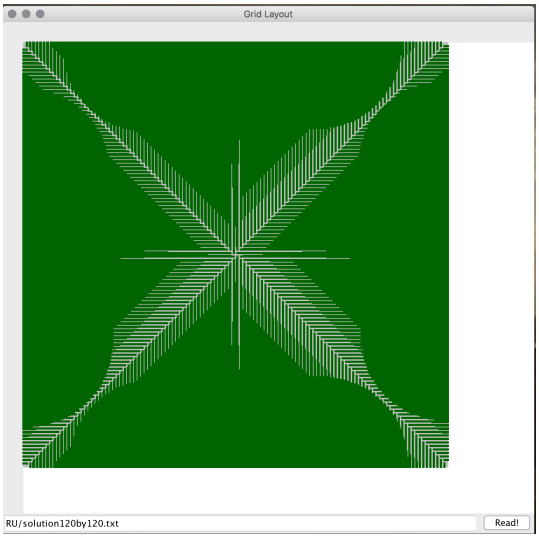
(a) 25 x 25



(b) 55 x 55



(c) 100 x 100



(d) 120 x 120

Figure 10: RuleRouter with different sizes

To sum up, the rule router routes the board in a quarter and uses successive path-finding for the terminals. Its main goal is to compress all the paths in a rectangle region (which is nearly visible in Figure 9) so that they do not cross the diagonal. The complexity is $O(l) = O(n^4)$, where l is the total length of the paths. So RuleRouter is significantly faster.

4 Result Analysis

4.1 Tables

The performance and correctness of different routers are shown in this section.

Comparison Between Routers

M	N	Gap Size	Flow	Length	Time(s)	Gap Size	Flow	Length	Time(s)	(cost_DC - cost_NF)/ cost_NF	time_NF / time_DC	Gap Size	Flow	Length	Time(s)	(cost_RU - cost_NF)/ cost_NF	time_NF/ time_RU
	NF					DC						RU					
5	5	1	25	79	4.16E-03	1	25	79	8.09E-04	0E+00	5.14E+00						
10	10	3	100	960	3.5E-01	3	100	960	2.04E-02	0E+00	1.72E+01	3	100	976	6.14E-04	1.64E-02	5.7E+02
15	15	4	225	3862	2.04E+00	4	225	3916	2.71E-01	1.38E-02	7.53E+00	4	225	4035	9.01E-04	4.29E-02	2.26E+03
20	20	6	400	11912	1.57E+01	6	400	11912	9.32E-01	0E+00	1.68E+01	6	400	12104	2.54E-03	1.59E-02	6.18E+03
25	25	7	625	27394	5.01E+01	7	625	27574	3.38E+00	6.53E-03	1.48E+01	7	625	28215	6.57E-03	2.91E-02	7.63E+03
30	30	8	900	55112	1.37E+02	8	900	55112	8.71E+00	0E+00	1.57E+01	8	900	55268	1.29E-02	2.82E-03	1.06E+04
35	35	10	1225	101775	5.05E+02	10	1225	102159	3.3E+01	3.76E-03	1.53E+01	10	1225	104015	3.13E-02	2.15E-02	1.61E+04
40	40	11	1600	170728	1.02E+03	11	1600	170728	5.17E+01	0E+00	1.97E+01	11	1600	171080	5.49E-02	2.06E-03	1.86E+04
45	45	13	2025	273183	2.79E+03	13	2025	273837	1.61E+02	2.39E-03	1.73E+01	13	2025	277901	1.18E-01	1.7E-02	2.36E+04
50	50	14	2500	411828	5.17E+03	14	2500	411828	2.78E+02	0E+00	1.86E+01	14	2500	412356	1.85E-01	1.28E-03	2.79E+04
55	55	16	3025	602856	1.2E+04	16	3025	603858	7.58E+02	1.66E-03	1.58E+01	16	3025	611409	3.65E-01	1.4E-02	3.29E+04
60	60	17	3600	847148	1.83E+04	17	3600	847148	1.09E+03	0E+00	1.68E+01	17	3600	848020	6.04E-01	1.03E-03	3.03E+04
65	65					19	4225	1168532	2.49E+03			19	4225	1181147	9.66E-01		
70	70					20	4900	1560480	3.62E+03			20	4900	1561636	1.29E+00		

As is shown in the table, the three routers all have the same solution k for the testcases.

The optimal length is obtained by DCRouter if n is even, and for odd numbers the length is about 0.1 % longer than optimal, but DCRouter is about 18 times faster.

For RuleRouter, the length is about 1 % longer for odd n, and about 0.1 % longer for even n. However, Rule Router is about 30000 times faster than NFRouter. It uses about 1 second for 65×65, which would costs more than 10 hours for NFRouter.

Table 1: Several other outputs by RuleRouter

M	N	Gap Size	Flow	Length	Time(s)
13	13	4	169	2571	0.00084
19	19	5	361	9720	0.001834
25	25	7	625	28215	0.006566
31	31	9	961	65312	0.019445
37	37	11	1369	130573	0.043896
43	43	12	1849	230801	0.083489
49	49	14	2401	386983	0.168701
71	71	21	5041	1676976	1.78427
77	77	22	5929	2296487	2.22882
83	83	24	6889	3095781	3.49984
89	89	26	7921	4087383	5.15996
95	95	28	9025	5299901	7.60512
101	101	29	10201	6730017	9.92689
107	107	31	11449	8471288	13.9473
113	113	33	12769	10530307	20.0613
119	119	35	14161	12943562	27.9761
125	125	37	15625	15749525	37.493

4.2 Validator

The validator can be used to verify correctness of results.

```
checking output solution50by50.txt...
Test Passed at type RU,case solution50by50.txt

checking output solution55by55.txt...
Test Passed at type RU,case solution55by55.txt

checking output solution60by60.txt...
Test Passed at type RU,case solution60by60.txt

checking output solution65by65.txt...
Test Passed at type RU,case solution65by65.txt

checking output solution70by70.txt...
Test Passed at type RU,case solution70by70.txt

Inconsistent gaps! N = 6 with 1 , 0 , 0
Inconsistent gaps! N = 7 with 2 , 0 , 0
Inconsistent gaps! N = 8 with 2 , 0 , 0
Inconsistent gaps! N = 9 with 2 , 0 , 0
Inconsistent gaps! N = 23 with 6 , 6 , 7
Inconsistent gaps! N = 33 with 9 , 9 , 10
Inconsistent gaps! N = 47 with 13 , 14
[matt154@Hongde-MacBook-Pro:Validator] $
```

Figure 11: Validator (DC, Rule routers not run for $N < 10$)

As is shown by the validator, all the solutions are valid solutions. However, it turns out that our Rule Router does not always output the minimum k (sometimes off by 1). This phenomenon is rare and occurs only 3 times among all the testcases.

5 Conclusion

As is illustrated in the text, rule based routing is significantly faster than optimal, network flow based routers. Though suboptimal (about 0.5 % errors on average), the router can be of great use due to its good efficiency and low error rate.

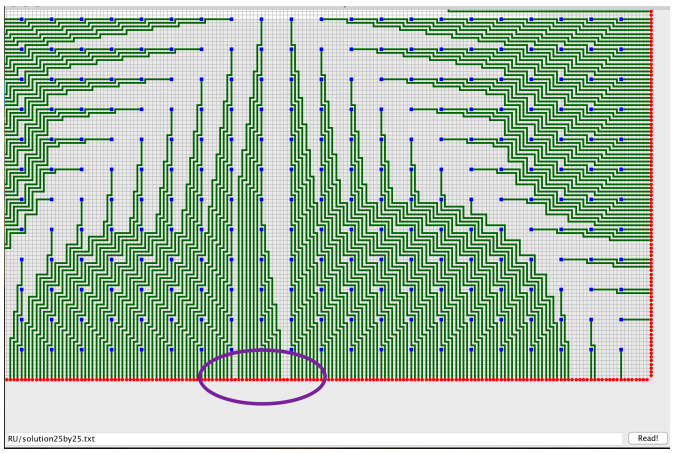


Figure 12: Unused terminals

We also want to note that there are still room for improvement. In particular, we should try to fix the center point issue when n is odd (mentioned in Section 3.2, also occured in RuleRouter), which may fix the error in k, and bring the 1% down to 0.1% for odd n. Also, we notice that there are unused terminals near center axis on the

Rule-based solutions. If we do more case analysis (which should require *really* delicate, tedious and long codes), we should be able to improve solution's quality.

References

[1] *Flow, on Algorithm Notes*. URL: http://www.csie.ntnu.edu.tw/~u91029/Flow2.html.