

# Flow Free Game Variant Solved Using Z3 SAT optimizer

Hong Wenhao, 2016011266

May 28, 2017

# Contents

- 0 Build, Run and Test 2**
  - 0.1 Building and Running . . . . . 2
  - 0.2 Testing . . . . . 2
- 1 Overview 5**
- 2 Data representation 6**
  - 2.1 Board representation . . . . . 6
  - 2.2 Internal Objects and Data . . . . . 6
- 3 Formulation to SAT 6**
  - 3.1 Overview . . . . . 6
  - 3.2 Variables Definition . . . . . 7
  - 3.3 Constraints Definition . . . . . 7
  - 3.4 Special Constraints: Optimization Goals . . . . . 7
  - 3.5 Pitfalls . . . . . 8
  - 3.6 Result Conversion . . . . . 8
- 4 Statistics 8**
- 5 Conclusion 11**

## 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 cases, including a built visualizer (written in Java Swing), are stored in this directory.

### 0.1 Building and Running

There are several tasks a user may wish to perform. Several makefiles are provided for this purpose.

*The makefile are written for Unix-like systems. You may need to modify the makefile in order for them to function under Windows.*

**Generator** To use the generator, run the following commands under the directory `/src`.

```
1 make generate
```

The generator produce 10 input files in the valid format in the directory `/testcase/set0/` with file names `0.txt`, `1.txt`, ... Note that, for the generator to function properly, `/testcase/set0` should exists.

**Solver** To use the solver, run the following commands under the directory `/src`.

```
1 make test
```

The solver produces several verbose output when running. The behavior of solver will be further described in 0.2.

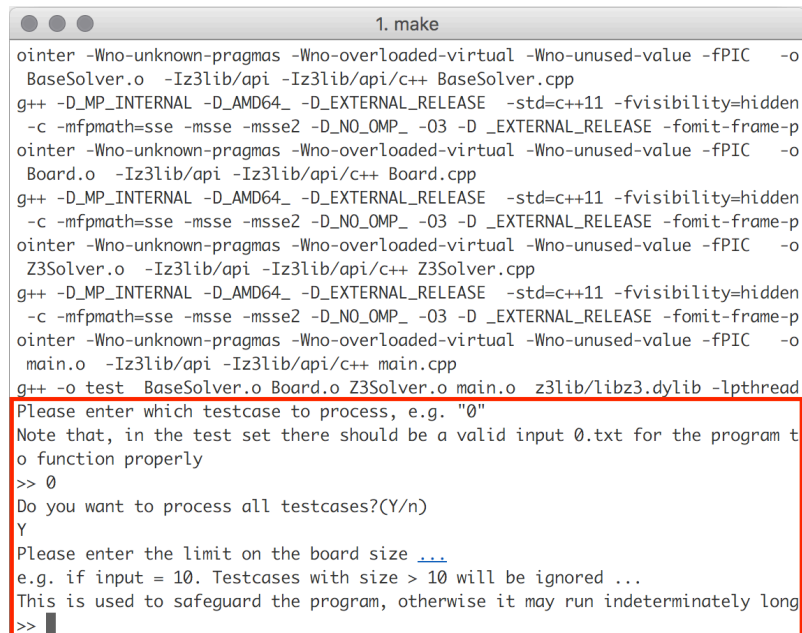
**Visualizer** The Visualizer was written in Java and is built beforehand. It can be found under the directory `/testcase`. However, you may wish to build the visualizer yourself. To build the visualizer, run the following command under the directory `/src/visualizer`.

```
1 make build_
```

Again, the behavior will be described in 0.2.

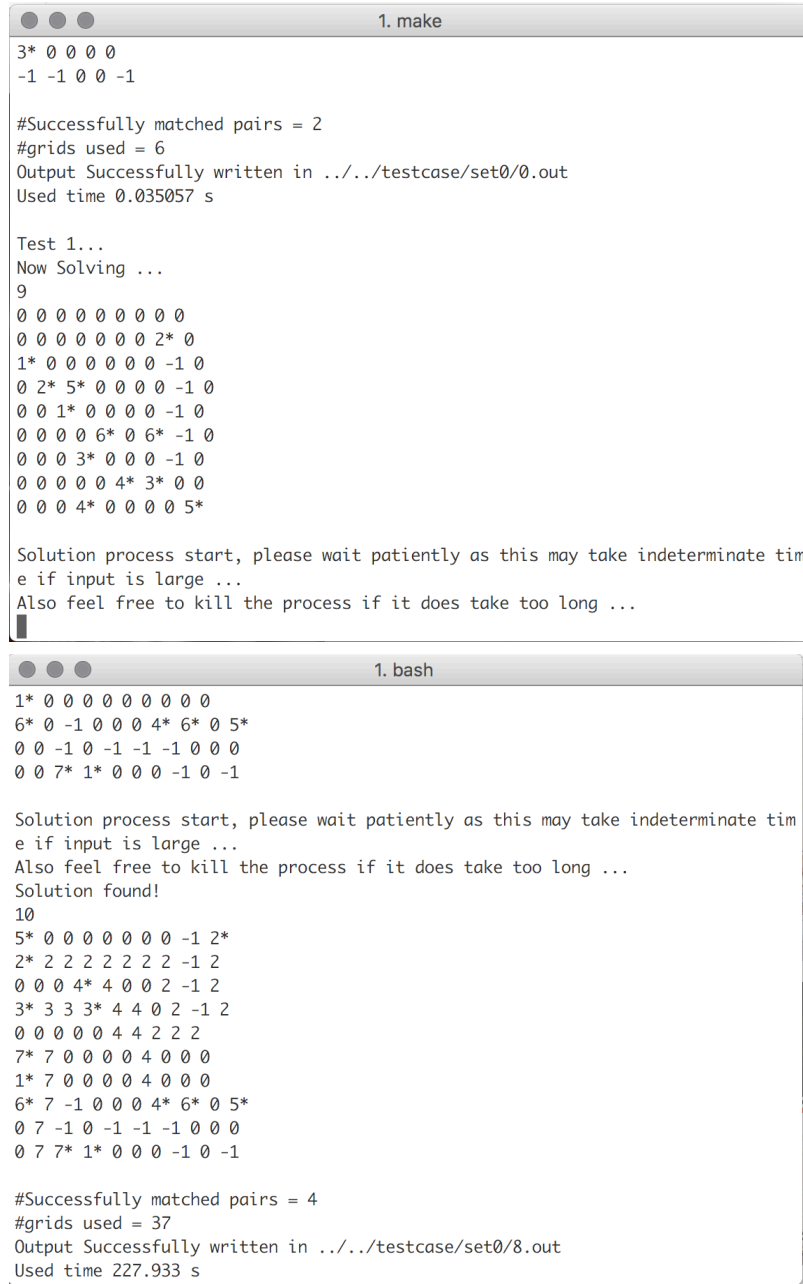
### 0.2 Testing

**Solver** After running the command in 0.1, the program `test` is runned in the terminal. First, it prompts the user to input which test case to run on. You should enter a single number in the range 0 ... 8.



```
1. make
ointer -Wno-unknown-pragmas -Wno-overloaded-virtual -Wno-unused-value -fPIC -o
BaseSolver.o -Iz3lib/api -Iz3lib/api/c++ BaseSolver.cpp
g++ -D_MP_INTERNAL -D_AMD64_ -D_EXTERNAL_RELEASE -std=c++11 -fvisibility=hidden
-c -mfpmath=sse -msse -msse2 -D_NO_OMP_ -O3 -D _EXTERNAL_RELEASE -fomit-frame-p
ointer -Wno-unknown-pragmas -Wno-overloaded-virtual -Wno-unused-value -fPIC -o
Board.o -Iz3lib/api -Iz3lib/api/c++ Board.cpp
g++ -D_MP_INTERNAL -D_AMD64_ -D_EXTERNAL_RELEASE -std=c++11 -fvisibility=hidden
-c -mfpmath=sse -msse -msse2 -D_NO_OMP_ -O3 -D _EXTERNAL_RELEASE -fomit-frame-p
ointer -Wno-unknown-pragmas -Wno-overloaded-virtual -Wno-unused-value -fPIC -o
Z3Solver.o -Iz3lib/api -Iz3lib/api/c++ Z3Solver.cpp
g++ -D_MP_INTERNAL -D_AMD64_ -D_EXTERNAL_RELEASE -std=c++11 -fvisibility=hidden
-c -mfpmath=sse -msse -msse2 -D_NO_OMP_ -O3 -D _EXTERNAL_RELEASE -fomit-frame-p
ointer -Wno-unknown-pragmas -Wno-overloaded-virtual -Wno-unused-value -fPIC -o
main.o -Iz3lib/api -Iz3lib/api/c++ main.cpp
g++ -o test BaseSolver.o Board.o Z3Solver.o main.o z3lib/libz3.dylib -lpthread
Please enter which testcase to process, e.g. "0"
Note that, in the test set there should be a valid input 0.txt for the program t
o function properly
>> 0
Do you want to process all testcases?(Y/n)
Y
Please enter the limit on the board size ...
e.g. if input = 10. Testcases with size > 10 will be ignored ...
This is used to safeguard the program, otherwise it may run indeterminately long
>> █
```

Then it asks whether you need to process all the 10 files. If yes, a limit number is needed (otherwise the program may not terminate when the board is large).



```

1. make
3* 0 0 0 0
-1 -1 0 0 -1

#Successfully matched pairs = 2
#grids used = 6
Output Successfully written in ../../testcase/set0/0.out
Used time 0.035057 s

Test 1...
Now Solving ...
9
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 2* 0
1* 0 0 0 0 0 0 0 -1 0
0 2* 5* 0 0 0 0 -1 0
0 0 1* 0 0 0 0 -1 0
0 0 0 0 6* 0 6* -1 0
0 0 0 3* 0 0 0 -1 0
0 0 0 0 4* 3* 0 0
0 0 0 4* 0 0 0 0 5*

Solution process start, please wait patiently as this may take indeterminate time if input is large ...
Also feel free to kill the process if it does take too long ...

1. bash
1* 0 0 0 0 0 0 0 0 0 0
6* 0 -1 0 0 0 4* 6* 0 5*
0 0 -1 0 -1 -1 -1 0 0 0
0 0 7* 1* 0 0 0 -1 0 -1

Solution process start, please wait patiently as this may take indeterminate time if input is large ...
Also feel free to kill the process if it does take too long ...
Solution found!
10
5* 0 0 0 0 0 0 0 -1 2*
2* 2 2 2 2 2 2 2 -1 2
0 0 0 4* 4 0 0 2 -1 2
3* 3 3 3* 4 4 0 2 -1 2
0 0 0 0 0 4 4 2 2 2
7* 7 0 0 0 0 4 0 0 0
1* 7 0 0 0 0 4 0 0 0
6* 7 -1 0 0 0 4* 6* 0 5*
0 7 -1 0 -1 -1 -1 0 0 0
0 7 7* 1* 0 0 0 -1 0 -1

#Successfully matched pairs = 4
#grids used = 37
Output Successfully written in ../../testcase/set0/8.out
Used time 227.933 s

```

The statistics are written in `/testcase/seti/stats.csv` where *i* is the number of test set.

A script `/testcase/collect.py` is provided to collect all the `stats.csv` from the test sets (0 to 8) into a single file.

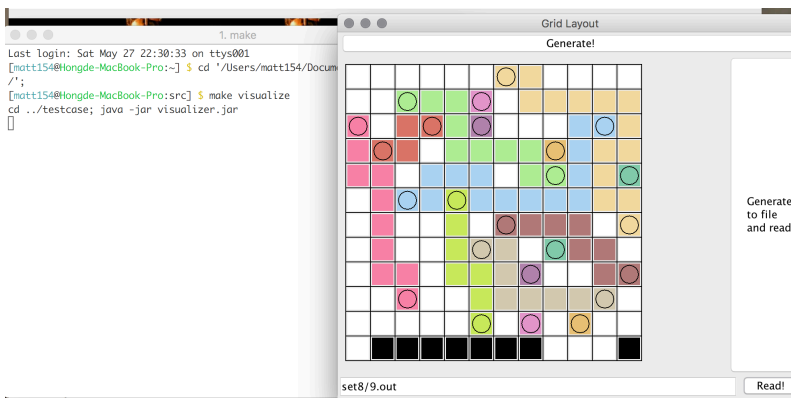
**Visualizer** The visualizer display the board using random colors. There are three buttons on the visualizer.



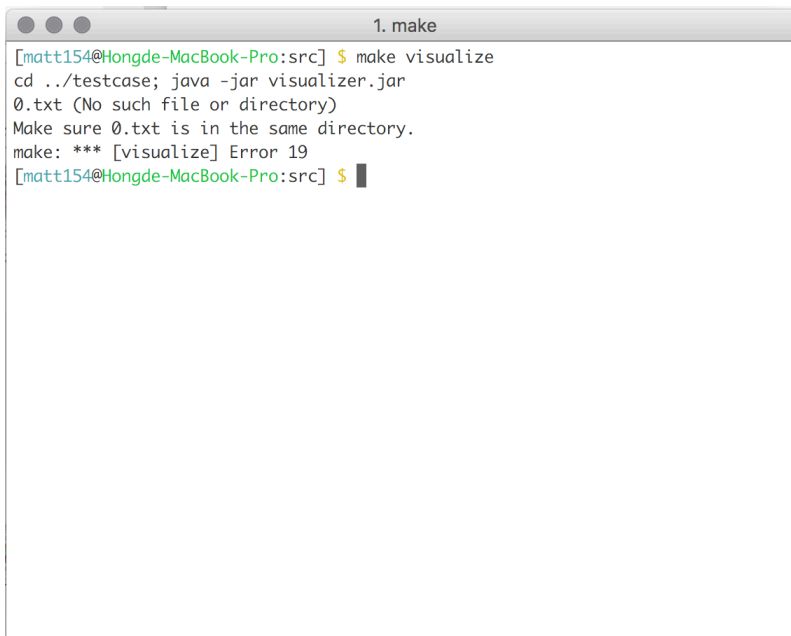
If generate is clicked, the visualizer generate a random layout of the board (which may not be valid input), this function is mainly for testing the visualizer.

Generate to a file and read has mainly the same function as “generate”, while it save the layout of the grid in a file named “grids.txt”.

To read a file which is in a valid format (see 2.1), input the file name in the text box at the bottom, and click read. You may want to click the button multiple times before a good looking color is gained.



When errors occur, the messages are printed on standard error, so it is recommended to used terminal to run the visualizer.

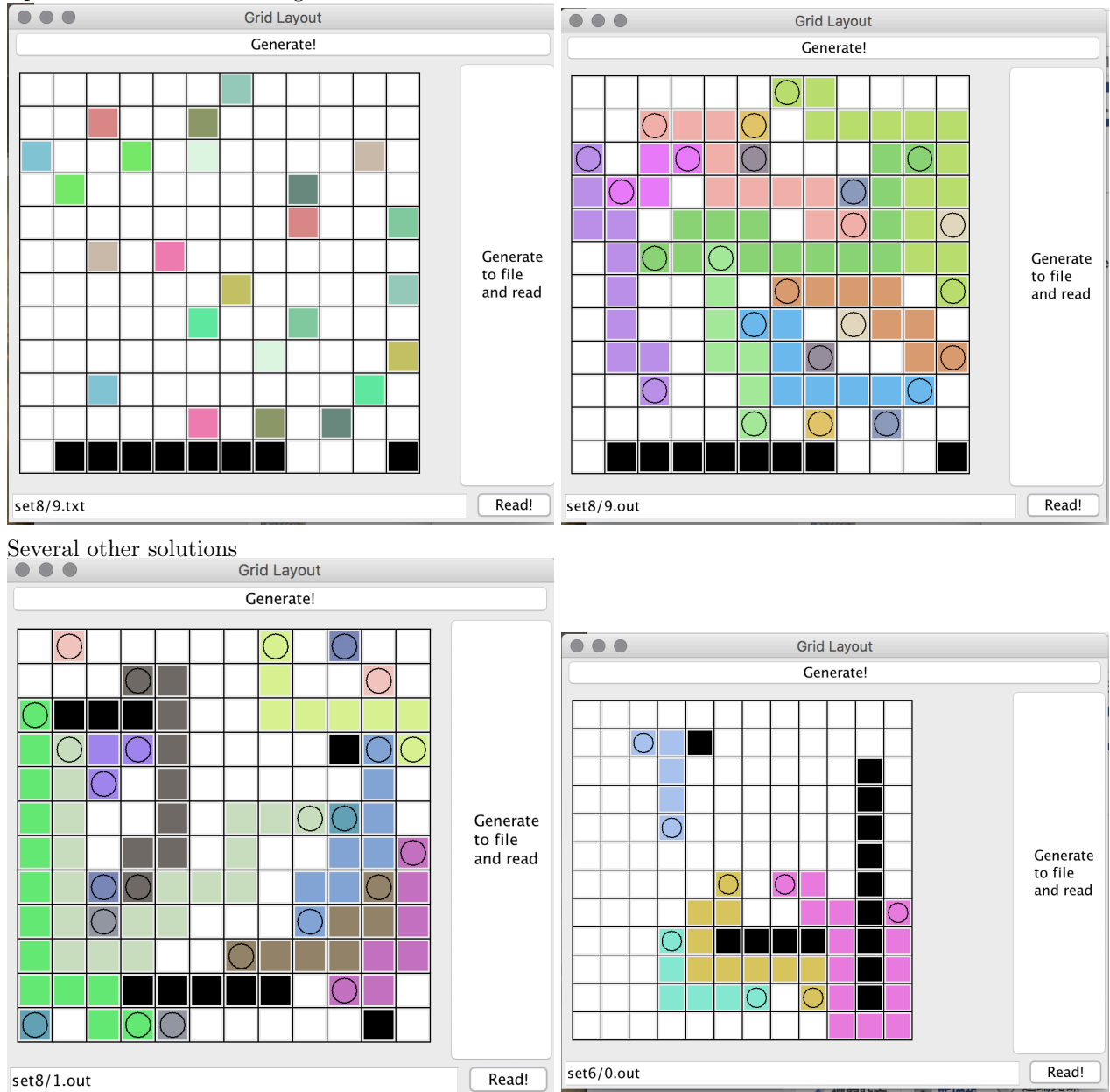


# 1 Overview

In this project a problem akin to the Flow Free game is solved. Given an  $n \times n$  grid along with some terminal points. The goal is to connect all the terminals using paths, which satisfy

1. Different paths cannot intersect.
2. Paths cannot contain obstacles (black). Note that terminals are also treated as obstacles of other colors.

There can be multiple solutions. Our purpose is to find a solution with the maximal pairs of terminals connected, and if tied, minimizing on the number of routing grids used. See the following layout for reference. On the right is the optimal solution our solver gives.



The projects' solution is based on a formulation to SAT problem, like that provided in [1], which is subsequently solved by the Z3Opt SAT optimizer by Microsoft Research. However, it should be noted that the example problem in [1] is much more complex than what we are trying to solve. So the resultant formulation in this project is much simpler.

## 2 Data representation

### 2.1 Board representation

One of the key design on data representation in the project is how to define a board layout. For this purpose, we give a lightweight representation of the board layout as defined in the following.

1. The first line contains an integer  $N$ , which is the dimension of the square board.
2. The following  $N$  lines, each containing  $N$  numbers, represent the contents on each grid of the  $N \times N$  board. Each number is either

- a) 0, which means an empty grid.
- b) -1, which means an obstacle grid.
- c) A number from  $1, 2, \dots, C$ , where  $C$  is the number of colors on the grid.

If the file is a valid input, these numbers should come in pair as they represent terminals.

If the file is a valid output, colors signify which terminal pair is using that grid. In this case, in order to distinguish terminals, the terminals are appended with an asterisk(\*) .

This format is used by the solver to receive input, and by the visualizer to draw the layout. The generator also generates a valid input in this format.

### 2.2 Internal Objects and Data

**Board** Internally, a board's information is saved in a Board object. It keeps two matrices, one keeping the color of each grid, the other keeping track of whether the grid is a terminal. It also keeps a set of Grid Object (which is an inner class representing a grid on the board) which are the terminals on the board.

The Board's constructor receives a file name and reads data formatted in 2.1 from the file. It also does some basic validating to prevent some invalid input.

#### Solver

**Base Solver** This is an abstract class of a solver. The interface of a solver, and most I/O are implemented in this base class. It internally keeps pointers to input Board and output Board to manipulate the solving process.

**Z3 Solver** This is an implementation of Base Solver. In particular, Z3 Solver handles the solution process using Z3Opt as engine. Internally it holds a bunch of helper methods to make the *void solve()* process easier to understand.

## 3 Formulation to SAT

In this section, the process of reducing the problem to SAT is described.

### 3.1 Overview

The key step, as illustrated in [1], is to determine the variables and constraints of the problem formulation. After setting these constraints to the solver, what is left is to trigger the SAT solution process, patiently wait for it to terminate, and decode the result back into our own problem. As the real solution process is handled by Z3, if the input is large the program is likely to run for several hours. Therefore a safeguard limit is needed by the solver to determine which cases to ignore when batch processing.

### 3.2 Variables Definition

*Side note* A straightforward formulation is to assign  $C$  boolean values,  $color_{i,j,c}$ , to each grid, where  $1 \leq i, j \leq N$  and  $0 \leq c \leq C$ , to represent whether the grid  $[i, j]$  is in color  $c$ . However, this would produce  $O(2^C)$  combination and might enlarge the search space, therefore a formulation to integer values are used instead. Hopefully this would reduce the search space (for each grid) to  $O(C)$ .

For each grid  $[i, j]$ , assign an integer variable  $color_{i,j}$ , representing the color on the grid. We restrict these variable to  $\{-1, 0, \dots, C\}$ , which has the same meaning as in 2.1.

Another helper function,  $routed : int \rightarrow Bool$  is defined for the formulation. This function keeps track of which colors are successfully matched, and the input range is normally  $\{1, 2, \dots, C\}$ . (We say *normally* because Z3 functions are total, they can receive other input, but the output are not specified in our formulation).

This formulation totally makes up  $O(C^{N^2} 2^C)$  search space, which is daunting for a direct search. However, we have not put our constraints yet. This constraints can significantly reduce the search space and, as Z3Opt is specifically designed for SAT problems, it is able to solve these problems to the scale  $N \approx 12$  and  $C \approx 12$ .

### 3.3 Constraints Definition

Without constraints, the solution given by Z3Opt is of no use. Several constraints are passed to Z3Opt to get usable results.

1. If the original board is empty, then assert  $0 \leq color_{i,j} \leq C$ . If the board is already an obstacle, assert  $color_{i,j} = -1$ . If the board is a terminal with color  $c$ , assert  $color_{i,j} = c$ .

This constraints is used to make the solution consistent to the input board.

2. For any grid  $color_{i,j}$  in the solution, assert the following properties.

- a. If  $color_{i,j} = 0$ , which means the grid is empty, do nothing
- b. If  $color_{i,j} = c$ , which means the grid is used as routing grid, then  
If  $!routed(c)$ , do nothing.

Else  $routed(c)$ , which means this color is routed, let  $countNear(i, j)$  to be the number of neighboring grids in the same color with  $color_{i,j}$ , assert  $countNear(i, j) = 1$  if the grid is a terminal, otherwise assert  $countNear(i, j) = 2$ .

Note that, by “neighbor”, we mean the grids sharing an edge with the grid of concern.

This constraint seems hard to understand. What it means is essentially that, any grid used as routing grid should have two neighbors. This property is an important feature of a valid solution. However, this is not enough to guarantee a valid solution as explained in 3.4, 3.5.

### 3.4 Special Constraints: Optimization Goals

For a board layout, there are be several colutions, which may not be optimal. Fortunately Z3Opt provides an optimization solver which can not only solve the problem, but also find a solution satisfying multiple optimization goals.

1. Maximize  $\sum_{c=1}^C routed(c)$ .

This constraint, in other words, tells the solver to maximize the number of successfully matched pairs.

2. Minimize  $\sum_{1 \leq i, j \leq n} (color_{i,j} \neq 0)$ .

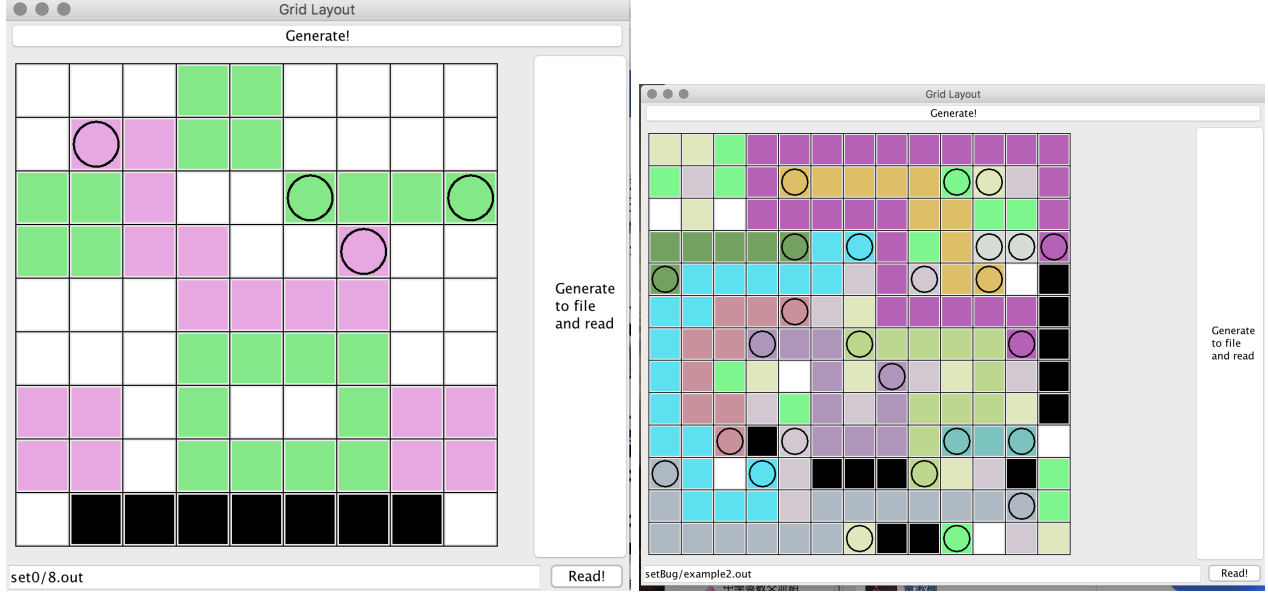
This constraint tells the solver to minimize the number of routing grids used. Here I do not take out the blocks and unrouted terminals because they do not make a difference in the optimal solution.

*This constraint is particularly important, deleting this constraints will yield invalid solutions as illustrated in 3.5.*



### 3.5 Pitfalls

If we omit the second optimization goal, we may get the following invalid results.



On the left, there are some mysterious “Cycles”. On the right, unrouted colors can appear randomly on empty grids.

It is apparent that the two solutions satisfy constraints in 3.3 but are not valid solutions. However, we can see that any optimal solution produced by our solvers are valid, because it also minimizes the number of routing grids used. After this elimination, the cases happening in the two pictures should not appear.

### 3.6 Result Conversion

After having Z3Opt solve the formulated problem, we obtain a so-called optimization model. We can further evaluate  $color_{i,j}$  and  $routed()$  to get our desired results.

## 4 Statistics

In this section we will examine several random test cases generated by the generator and see our solver’s performance and correctness.

Table 1: Set 0 (dense (more terminals) and small board)

Board size	Number of terminals	Number of matched pairs	Number of grid used	used time (s)
5	3	2	6	0.038952
9	6	5	26	19.3948
7	4	3	23	0.225982
9	6	4	36	20.9906
9	6	4	32	0.681243
9	6	5	46	11.8672
7	4	3	18	0.181918
8	5	4	10	0.30254
10	7	4	37	206.552
7	4	3	19	1.03339

It can be seen that running time is very small if  $n$  is small. When  $n$  reaches 9 or 10 the test seems to be slower. It should be noted that in test 5, the running times is very small even if  $n = 9$ .

Table 2: Set 1 (small board, more obstacles)

Board size	Number of terminals	Number of matched pairs	Number of grid used	used time (s)
10	5	4	35	130.954
8	4	3	27	0.341012
7	3	3	12	0.07255
11	5	NAN	NAN	NAN
11	5	NAN	NAN	NAN
7	3	3	15	0.135479
9	4	4	25	4.79281
6	3	2	9	0.060629
6	3	3	12	0.071265
6	3	1	7	0.032811

In this set, two test cases are abandoned because they are likely to run for a *very* long time. It can be seen that, with more obstacles the overall runtime is shorter compared to Figure 1. This is because the search space is reduced.

Table 3: Set 2 (small board, less obstacles)

Board size	Number of terminals	Number of matched pairs	Number of grid used	used time (s)
9	4	4	34	3.44555
6	3	3	9	0.048485
8	4	4	20	1.19083
8	4	4	26	1.90143
7	3	3	18	0.23966
9	4	4	38	72.605
11	5	NAN	NAN	NAN
9	4	4	20	7.72392
7	3	3	21	0.763993
11	5	NAN	NAN	NAN

In this set, the running time is slightly longer because there are less obstacles.

Table 4: Set 3 (small board, dense terminals)

Board size	Number of terminals	Number of matched pairs	Number of grid used	used time (s)
8	4	4	19	0.368898
5	2	2	9	0.031933
6	3	2	9	0.015894
8	4	3	29	34.4024
8	4	3	16	1.76329
9	4	4	34	0.712702
7	3	3	14	0.404006
5	2	2	6	0.015825
5	2	2	5	0.012902
6	3	3	12	0.054895

As the board become denser, the search space is reduce. So the overall performace is improved.

Table 5: Set 4 (larger board, 1-3 terminals)

Board size	Number of terminals	Number of matched pairs	Number of grid used	used time (s)
10	2	1	9	8.28883
11	2	2	14	1.34059
11	2	2	12	1.54678
11	2	2	15	3.15044
10	2	2	20	0.964794
9	1	1	6	0.046412
12	2	2	22	10.0036
11	2	2	15	1.01513
9	1	1	14	0.072449
11	2	2	17	3.46671

If the board is large and there are sparse terminals, the search space becomes significantly larger. However, our solver can still terminate if the terminals come in 1 to 3 pairs.

Table 6: Set 5 (larger board, 1-3 terminals)

Board size	Number of terminals	Number of matched pairs	Number of grid used	used time (s)
10	2	2	7	1.33025
9	2	2	11	0.183871
8	2	2	26	0.12793
11	2	2	13	0.766032
11	2	2	17	4.51389
10	2	2	15	1.158
12	3	3	17	1.5079
13	3	NAN	NAN	NAN
9	2	2	11	0.477976
10	2	2	17	0.437725

This is the same type of data as Set 4.

Table 7: Set 6 (general)

Board size	Number of terminals	Number of matched pairs	Number of grid used	used time (s)
12	4	4	35	20.958
13	4	NAN	NAN	NAN
8	2	2	6	0.100144
9	3	3	29	11.9208
9	3	3	14	0.16763
8	2	2	10	0.239685
10	3	3	17	5.70701
13	4	NAN	NAN	NAN
13	4	NAN	NAN	NAN
10	3	3	17	3.88999

Our solver works well with general test cases which are medium in size, and contain 3 to 6 terminal pairs.

Table 8: Set 7 (general)

Board size	Number of terminals	Number of matched pairs	Number of grid used	used time (s)
6	4	4	28	0.093502
5	3	2	9	0.021829
9	6	4	30	1.85562
9	6	4	33	0.748592
10	6	6	43	39.0728
10	6	5	30	25.2425
10	6	4	41	11.0599
7	4	4	23	0.20926
7	4	3	14	0.377056
5	3	2	14	0.023671

Our solver works well with general test cases which are medium in size, and contain 3 to 6 terminal pairs.

Table 9: Set 8 (devil’s test case, large and dense)

Board size	Number of terminals	Number of matched pairs	Number of grid used	used time (s)
13	13	10	96	141.198
12	12	8	73	339.962
10	10	6	39	67.0031
10	10	5	39	1.93291
15	15	NAN	NAN	NAN
15	15	NAN	NAN	NAN
10	10	7	50	3.84604
12	12	3	40	142.191
10	10	5	24	0.951619
12	12	8	75	1356.63

For large board and  $O(n)$  pairs of terminals, our solver perform worse but can still terminate (compared with set 5 and set 6). This is again, because the search space is reduced.

## 5 Conclusion

After our experiments, we found that the problem can, indeed, be formulated into a satisfiability problem and can be solved using Z3Opt. This kind of technique is powerful and can be applied to a wide range of chip related problems as mentiond in [1]. Its advantage is that it is a general problem solving method and is guaranteed to be optimal. On the other hand, though, I believe heuristics results are still important because in many cases, suboptimal results are sufficient.

## References

- [1] Oliver Keszocze et al. “A General and Exact Routing Methodology for Digital Microfluidic Biochips”. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '15. Austin, TX, USA: IEEE Press, 2015, pp. 874–881. ISBN: 978-1-4673-8389-9. URL: <http://dl.acm.org/citation.cfm?id=2840819.2840941>.