An Implementation of SAT-Based Two-Terminal Path Finding Using Z3 Solver

YiHong Gu
Tsinghua University
Beijing, China
gyh15@mails.tsinghua.edu.cn

1 Introduction

The 'Two-Terminal Path Finding Problem' has the following description: given $N \times N$ routing grids, and M pairs of terminals, you should use program to find routing paths connecting each pair of terminals. The limit is that different paths cannot cross each other, and no path can cross obstacles. The objective is to maxmize the number of connected pairs, and when all pairs of terminals can be connected, you should minimize the total length of all the paths.

In order to make the objective more clear and not change the original idea of the problem, we simply make the following stipulates that if one grid is defined to be one terminal of pair d, then other pairs $d'(d \neq d')$ can not route across the grid.

2 Related Work

[1] gives a general routing methodology which mainly uses 3 kinds of variable as parameters and use Z3 solver to solve the problem. The abstract idea is that giving a concrete illustration of each pairs: for each pair, define a droplet that starts from the origin and ends in terminal during time t^* and $t^\dagger$, and then define the binary variable $c^t_{p,d}$ to illustrate whether the droplet d is at the position p in time t. It sets a lot of limits to ensure that (1) each droplet d appears in the board during time t^*_d to time $t^\dagger_d$ and it is at its origin position p^*_d in time t^*_d and at its terminal position $p^\dagger_d$ in time $t^\dagger_d$ (2) every second each droplet only appears at one position (3) the droplet satisfy the routing rule that can only step to one adjacent position per second.

Z3 solver [2], developed by Microsoft Research, can use optimized algorithm to check whether it can satisfy all constraints and give a concrete solution. Moreover, it can also maximize/minimize one single variable or expression under all constraints.

3 SAT-Based Model

This part is divided into three main parts, sperately illustrate three different models: the original model which resembles the model established in [1], the model which resembles network flow, the model which combined the 2 models above.

In the beginning, we just emphasize the definitions:

- N: the size of the routing boards.
- M: number of pairs.
- d: the index of one single droplet.
- t: the time.
- p: one single position, can be represented by $p = (x, y) \in \{1, \cdots, N\} \times \{1, \cdots, N\}$.
- P: the set of all p.
- $B(p)$: the set of positions that are adjacent to the position p in four direction(up, down, left, right).
- p^*_d, $p^\dagger_d$: the origin and terminal of droplet d.

3.1 Model 1: Based on the Paper

We use the notation that resembles the original notation in [1], and define the binary variable $c^t_{p,d}$ to illustrate whether the droplet d is at the position p in time t, and the constraints can be divided into four main parts:

to be or not to be: we can simplify the t^* and $t^\dagger$ and assume every droplet appears at the board in time 1 (if we want to connect the two-terminal pair), so the constraints can be (for each d):

$$c^1_{p^*_d,d} \rightarrow \bigvee_{t=1}^{T} c^t_{p^*_d,d}$$

It means: if it appears, then it must reach the terminal.

second conflict: it means every second each droplet only appears at one position, so the constraints can be (for each d):

$$\bigwedge_{t=1}^{T} \bigwedge_{p \in P} \bigwedge_{p' \in P, p' \neq p} \neg c^t_{p,d} \vee \neg c^t_{p',d}$$

position conflict firstly every grid can only be visited once (for each p)

$$\bigwedge_{d=1}^{M} \bigwedge_{d'=d+1}^{M} \bigwedge_{t=1}^{T} \bigwedge_{t'=t+1}^{T} \neg c^t_{p,d} \vee \neg c^{t'}_{p,d'}$$

$$\bigwedge_{d=1}^{M}\bigwedge_{t=1}^{T}\bigwedge_{t'=t+1}^{T} \neg c_{p,d}^{t} \vee \neg c_{p,d}^{t'}$$

Also, for each position p which has obstacles for droplet d, for all t, $c_{p,d}^{t}$ must be FALSE.

travel constraints: it means the droplet satisfy the routing rule that can only step to one adjacent position per second. So for each droplet d, if it appears at position p in time t, it must appears at the adjacent position p' in time $t-1$ (except $t=1$).

$$c_{p,d}^{t} \rightarrow (\bigvee_{p' \in B(p)} c_{p',d}^{t-1})$$

optimization object: if we want to maxmize the total pairs, we just need to maxmize

$$\sum_{d=1}^{M} [c_{p_d^*,d}^{1}]$$

If we want to minmize the total length, we firstly enforce that every $c_{p_d^*,d}^{1}$ should be TRUE and then minimize

$$\sum_{d}\sum_{p}\sum_{t} [c_{p,d}^{t}]$$

Note that the notation

$$[statement] = \begin{cases} 1, & statement = TRUE \\ 0, & statement = FALSE \end{cases}$$

We can find the scale of the variables and the contraints are very large, we have totally $T \times N^2 \times M^2$ variables and almost $O(T^2 N^4 M^2)$ contraints, which T represents the maximum time used and it can be $\frac{N^2}{2}$ under the worst condition. If we simply make N be 10, M be 4, we can't afford to work it out by our personal computer.

3.2 Model 2: Based on the Network Flow

In this model, we directly use the model resembles network flow architecture, for each droplet d, we build a network flow graph and set the source to be the origin and sink to be the terminal, and all the positions must satisfy the flow conservation. Then we have M layers network flow graph, afterwards we implement the constraint that every position can be only visited once using a constraint like linear constraint.

In this model, if we use some artifice, we can only have totally $2 \times N^2 \times M$ variables and $O(\times N^2 \times M)$ contraints. Besides, after a large quantity of attempts we find if we conbine several contraints together, we can accelerate the speed of our algorithm, and this things can be concretely described in the next section.

3.3 Model 3: The Mixed One

Combine model 1 and model 2 together, we can find a more efficient model, which only have $N^2 \times M$ variables and is 1.5 times faster than model 2.

We use the binary variable $c_{p,d}$ to represent whether the droplet d visit across the position p, therefore the contraints are as follows:

limit terms if it is a barrier or is not in the corresponding layer, we enforce $c_{p,d}$ to be FALSE, also we force that every position can only be visited once by the following contraint (for each p)

$$\sum_{d=1}^{M} [c_{p,d}] \leq 1$$

source and sink terms: for each d, we come up with a contraint that

$$c_{p_d^*,d} \leftrightarrow c_{p_d^\dagger,d}$$

connection terms: for each p which is not the terminal/origin/obstacle grid, it must satisfy the following contraint for each d:

$$c_{p,d} \rightarrow (\sum_{p' \in B(p)} c_{p',d} = 2)$$

and for each p which is equal to p_d^* or $p_d^\dagger$ for a specific d, it must satisfy the following contraint:

$$c_{p,d} \rightarrow (\sum_{p' \in B(p)} c_{p',d} = 1)$$

optimization object: if we want to maxmize the total pairs, we just need to maxmize

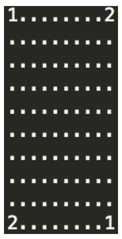
$$\sum_{d=1}^{M} [c_{p_d^*,d}]$$

If we want to minmize the total length, we firstly enforce that every $c_{p_d^*,d}$ should be TRUE and then minimize

$$\sum_{d}\sum_{p} [c_{p,d}]$$

4 Optimization

However, the models above don't have efficiency when the graph is sparse. For example, for a board that $N=10$, if we set $M=9$ and place the 9 pairs over the whole board, it runs quite fast (about 1.5 second, model 2), but if we set $M=2$ and place the 2 pairs like the following Figure1, it takes almost 30 minutes to get the best answer.



```
1........2
..........
..........
..........
..........
..........
..........
..........
..........
2........1
```

Figure 1: An Extreme Example

In order to optimize the algorithm, we come up with three main points:

(1) We can combine some muti-SAT terms together to a mixed term, which can reduce the check time (it can approximately accelerate the speed twice).

(2) It has no meaning if one droplet d takes the squared area, it means the situation

$$c_{p_1,d} = c_{p_2,d} = c_{p_3,d} = c_{p_4,d} = TRUE$$

should be forbidden where $p_1 = (x, y)$, $p_2 = (x+1, y)$, $p_2 = (x, y+1)$, $p_4 = (x+1, y+1)$, Z3 solver will search all the posibilities including this situation. However, under the circumstances of this specific problem, this situation don't contribute any to the final answer. Therefore, we can manually add the **prune term 1** for each p and d (where $p_1 = p$):

$$\neg(c_{p_1,d} \wedge c_{p_2,d} \wedge c_{p_3,d} \wedge c_{p_4,d})$$

(3) when consider the 4 totally empty grid (no obstacle/terminal), for one single droplet, there are 2 ways to pass from the left-down to up-right, just like the Figure 2.

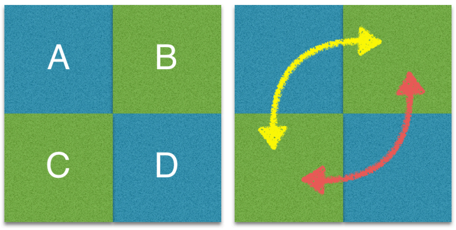


Figure 2: A 4-Block View

we can find the path C-A-B and C-D-B are both ok, and Z3 solver will check all the 2 path, so we can forbid this situation simply by the following contraint **prune term 2**(we forbid C-D-B for any p and d):

$$(\sum_{x=1}^{M} [c_{p_1,x}] = 0) \rightarrow \neg(c_{p_2,d} \wedge c_{p_3,d} \wedge c_{p_4,d})$$

where $p = p_1 = (x, y)$, $p_2 = (x+1, y)$, $p_2 = (x, y+1)$, $p_4 = (x+1, y+1)$. Moreover, we can strength the contraint by the following expression if we check other conditions.

$$(\sum_{x=1}^{M} [c_{p_1,x}] = 0) \rightarrow \neg(c_{p_2,d} \wedge c_{p_3,d})$$

and another symmetric terms should also be added (to forbid A-C-D from 2 paths A-C-D and A-B-D)

The prune term 1 and prune term 2 is of great significance, if we use these terms, we can get the answer of example in Figure 1 in 1.5 seconds.

5 Results and Evaluation

6 System Architecture

6.1 Overview

The whole system architecture can be divide into four main parts

- the core part, which consists of the whole algorithm(routingalgo.h, routingalgo.cpp, routing.h, routing.cpp, message.h, message.cpp, routing_test.cpp).
- the checker part, which is implemented to check if the answer calculated by the algorithm is legal and correct(checker.h, checker.cpp).
- the evaluation part, which is implemented to evaluate the efficiency and correctness of the algorithm(evaluate.h, evaluate.cpp, report.cpp).
- the demo part, we develop a simple demo, you can define your own maps in GUI and view the general results and detailed results to better understanding the problem from a dropout perspective (demo.cpp, demo.h, editor.cpp).

The makefile file can generate three different executable files

- **make test**: generate a executable file that can choose one/several algorithm to get a answer for a given testcase.
- **make repo**: generate a executable file that evaluates different algorithms.
- **make demo**: generate a executable file that gives a GUI, which you can define the map and run it using a concrete algorithm.

We will introduce the four main parts of our system in the following subsection.

6.2 Core

The CORE part use a design pattern of strategy, mainly consists of 3 main group of classes

- **RoutingSolver**: make preparations and call the main algorithm
- **RoutingAlgo** and the classes inherit it **RoutingSatAlgo**, **RoutingSatAlgoFlow**, **RoutingSatAlgoFlowPrune**, **RoutingSatAlgoPointPrune**: it is the main algorithm
- **message**: store the answer of the algorithm

Figure 3 shows the inheritance relationship of these classes (except class message).

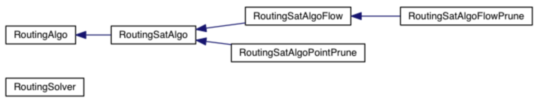


Figure 3: Inheritance Relationship in Core Part

RoutingSolver's UML diagram is in Figure 4, we can see that we must make one algorithm be the composition of RoutingSolver when a RoutingSolver object is created. the function of it's member functions are as follows:

- load() : load the vector of string from file
- getBlock() : get the map (stored as a matrix) from the file
- solver(): main function that load data from file and compute the answer by calling the corresponding algorithm.
- solve(): call the algorithm

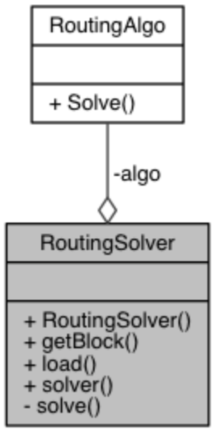


Figure 4: RoutingSolver's UML diagram

RoutingAlgo is an abstract class of the algorithm, **RoutingSatAlgo** inherit it and add some common member variables and member functions of sat model, here is the details:

member variable:

- sat_context, one, zero are variable for z3 solver. one and zero is the const expression of integer 1 and 0, just for convenience.
- n, m, D is the basic information of the board, separately represent the height, width and number of droplets of the board.

member funtcion:

- GetValue(): this is just an adaptor and converts the expression of z3 to a char ('t' or 'f').
- BoolToInt(): convert a bool expression in z3 to a int expression.
- GetStartEndPoints(): get the origin and terminal from the board.
- GetAns(): get the answer according to the arguments [aug](aug = 0 means get the maximum pairs, aug = 1 means get the minimum length).

We just introduce the member variable and member function of class **RoutingSatAlgoPointPrune** to introduce our structure of algorithm.

- idx(), getindx(): get the index of a binary variable(such as $c_{p,d}$), because we use z3::expr_vector(an 1-dim vector) to store the binary variables.
- AddSatTerms(): define binary variables of the model.
- EstablishModels(): define contraints of the model.
- PruneTerms(): add some prune contraints.
- FindPath(): find the final path via the results of z3.

The Figure 5 gives an overview of the family of algorithms.

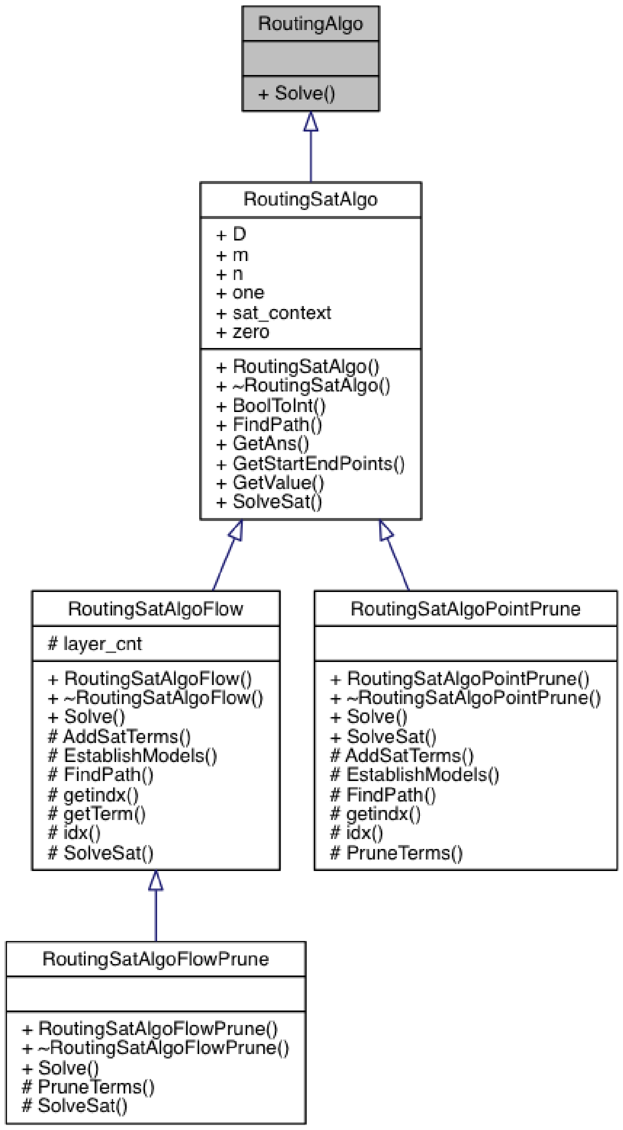


Figure 5: RoutingAlgo's UML diagram

When it comes to the concrete implement of our algorithm, we use the contraints which illustrated in Section 3 and try to combine several expressions to one single expression to reduce the time.

The class **message** store the answer of the algorithm.

- board, d_, m_, n_: store the main information of the original board.
- totlength, totpair: store the total length of the answer and total connected pairs of the answer.
- push(): store a new step of droplet d.
- display(): display the final board(consists of path) and the detailed paths of all connected pairs.

Figure 6 shows the details of the class.

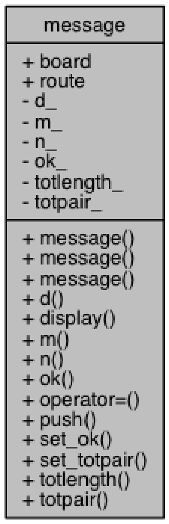


Figure 6: message's UML diagram

6.3 Checker

The Checker part consists of 2 single classes:

- **CheckerMessage**: store the detailed checker information.
- **AnswerChecker**: check if one answer is legal and correct.

In **CheckerMessage**, ok shows if the answer pass the check process and res shows the detailed information, Figure 7 shows concrete information.

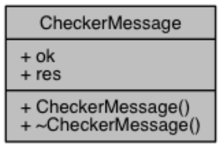


Figure 7: CheckerMessage's UML diagram

In **AnswerChecker**, we make 2 kinds of check processes.

- legal_check(): check if the given output is legal, we divided this processes into 3 parts, separately implemented in pairs_check() (check if the out.totpair is correct), length_check() (check if the out.totlength is correct), legal_step_check() (check if the step is legal, including checking the origin, terminal and every step).
- cmp_check(): check if the answer of [ans] is better than [out], we assume [ans] is a solution that is approximately best, if [out] is worse than [ans], [out] is not the best solution, detailed evaluation method will be illustrated in the next subsection.

Figure 8 shows an overview of class AnswerChecker.

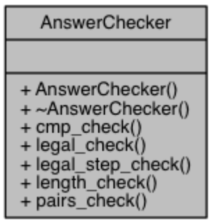


Figure 8: AnswerChecker's UML diagram

6.4 Evaluation

The Evaluation part consists of 3 main class families.

- **DataGenerator** and **RandomDataGenerator**: they are designed to generate test data.
- **Time**: it is designed to store the time used by a concrete algorithm and analyse in a naive method (calculate the average time used and the standard deviation) to give a summary of the efficiency of the algorithm.
- **Evaluation**: it is designed to evaluate the correctness and efficiency of the algorithm.

The class family **DataGenerator** uses the design pattern of strategy, **RandomDataGenerator** will generate a random test case if the size of the board (N), the number of droplets (M), the proportion of obstacles and the random seed are given, Figure 9 shows more information:

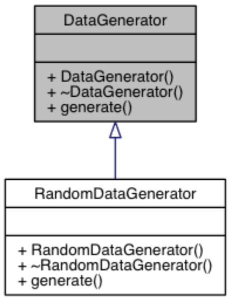


Figure 9: DataGenerator's UML diagram

By class **Time**, we can add a new time span (push_back()) and get the summary data(average(), standard_deviation()) easily, more infomation can be seen in Figure 10

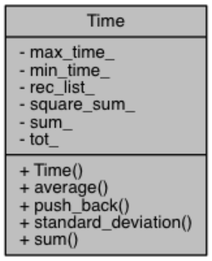


Figure 10: Time's UML diagram

Having the tools of data generator, time recorder and checker, we can evaluate our algorithm in the following way:

We can create an object of the class **Evaluation** to evaluate two specific algorithms by one kind of test data which generated by a specific data generator. We execute the evaluation several times. Therefore, when we create an object of Evaluation, we should give these arguments: algo-a, algo-b, data-generator, the number of test-cases in this evaluation.

For each test case, seed is different, but other arguments are same. After generating the test data, we execute the two algorithms separately and record their use of time, then we check the legality of the answers given by the two algorithms, finally we use cmp_check() in AnswerChecker to check if the answer given by one algorithm is worse than the answer given by another algorithm. Some people think this method might not be useful, but it really works. In our evaluation, even two algorithms all have mistakes, we can also detect the errors and finally fix the bugs. The communication diagram of class Evaluation in Figure 11 can give an overview:

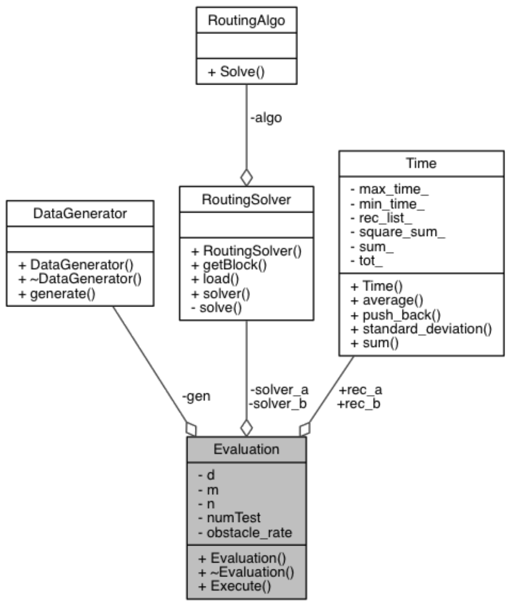


Figure 11: Evaluation's communication diagram

6.5 Demo

The demo part uses the design pattern of state and including 2 major class family.

- **State**, which is inherited by 5 classes, and the inheritance relationship is illustrated in Figure 12.
- **Demo**, uses class State, the state design pattern and a lot of corresponding member variables to give a GUI to the user.

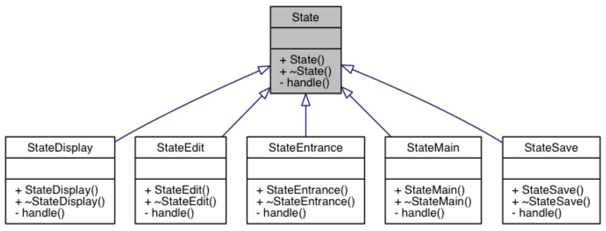


Figure 12: State's inheritance relationship diagram

As shown above, the class **State** is inherited by 5 classes, and the functions of the 5 classes are as followings.

- **StateEntrance**: the state of entrance interface, let user to define the size of the map.
- **StateMain**: the state of main interface, user can view the map, save the map, edit the map or let the algorithm calculate the answer.
- **StateEdit**: the state of edit interface, user can edit the map: move the cursor by w/a/s/d; enter '1' - '9' to add terminals; enter 'o' to add obstacles.
- **StateSave**: the state of save interface, user can enter the filename of the map
- **StateDisplay**: the state of display interface, user can view the answer and see how droplets path to get further understanding of the idea of droplet

We can see how the class **Demo** organize all the things together by seeing the following communication diagram of it in Figure 13

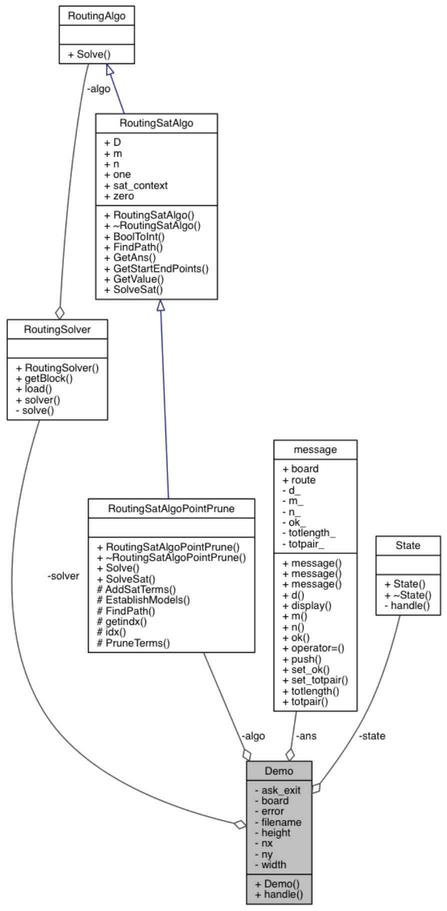


Figure 13: Demo's communication diagram

7 Data Flow

We describe out data flow using the example of 3 executable files shown in the previous section.

7.1 Test

In the beginning, the object of one particular routing algorithm (the class which inherit the class **RoutingAlgo**)is created, and we use it's address as a pointer to create the object of **RoutingSolver**.

Afterwards, we send a string to the object **RoutingSolver** and it calls the function load() to get the map, then it convert the map, which in form of vector<string>, to the objective form vector<vector<int> >, and use it, the size of the board (two integers) and the number of pairs (one integer) as parameters to call the main function Solve() in a concrete **RoutingAlgo** object.

The function Solve() firstly add an [aug] = 0 and calls the function SolveSat(), which means the object is to find the solution with largest number of pairs. In the main process of SolveSat(), we create expr_vector as binary variables and call the functions AddSatTerms(), EstabishModels(), PruneTerms() successively to build the model, it use GetAns() to check if it should calulate the minimum length (set [aug] = 1 and recall the function SolveSat()), and finally calls the function FindPath() to return a **message** as a result.

Therefore, it receives a **message** variable that returns the ans of the given map.

Afterwards, it uses function getBlock() in **RoutingSolver** to get the map in form of vector<vector<int> > and put the map(vector<vector<int> >) and the ans(message) as parameters to the function legal_check in the object of **AnswerChecker** to check if the ans is legal, after many turns of check, it returns an object of **CheckerMessage** as a result of checker.

7.2 Report

In the beginning, it creates the objects of two diffent implementation of the abstract class **RoutingAlgo** and one object of the implementation of the abstact class **DataGenerator**, and use the addresses of these objects as pointer to create an object of **Evaluation**. (Meanwhile, it should send the arguments that decide the type of the test data)

Afterwards, it calls the function execute() in the object of **Evaluation**. Therefore, for each test, it uses the arguments which describe the data type as parameter to call the data generator to generate one single test. Then it uses the way described in the previous subsection to get the two answer and check it, meanwhile it calculate the time they used separately and send it to the composition object **Time** in class **Evaluation**.

Finally, we can print out the detailed information stored in the object **Time**.

You can also refer to the communication diagram of class **Evaluation** for better understanding in Figure 11

7.3 Demo

The data flow is quite obvious in the part of demo and you can refer to both the communication diagram of class **Demo** in Figure 13 and the next section for better understanding.

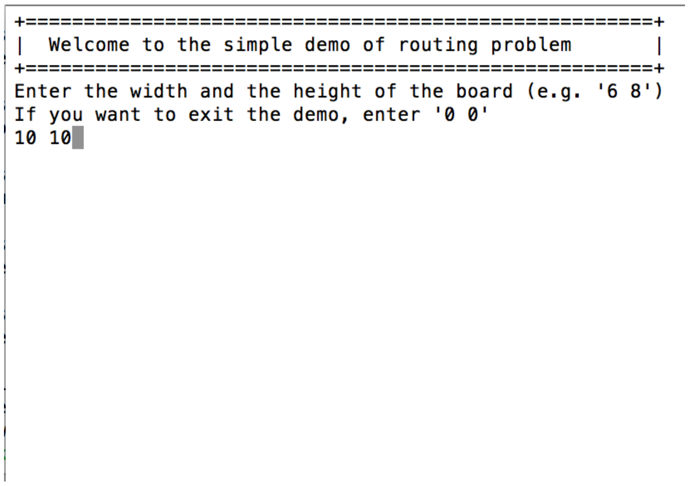
8 Tutorial for GUI

We also develop a GUI to let the user define their own map, and here is a simple tutorial.

Firstly, compile it and call the executable file.

```
$ make edit
$ ./editor
```

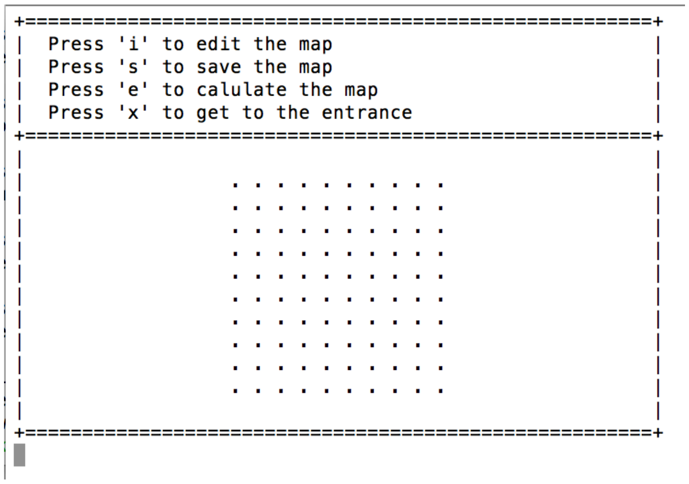
then you enter the entrance interface, you should enter the size of the board (like Figure 14):



```
+====================================================+
|  Welcome to the simple demo of routing problem      |
+====================================================+
Enter the width and the height of the board (e.g. '6 8')
If you want to exit the demo, enter '0 0'
10 10
```

Figure 14: GUI: Entrance Interface

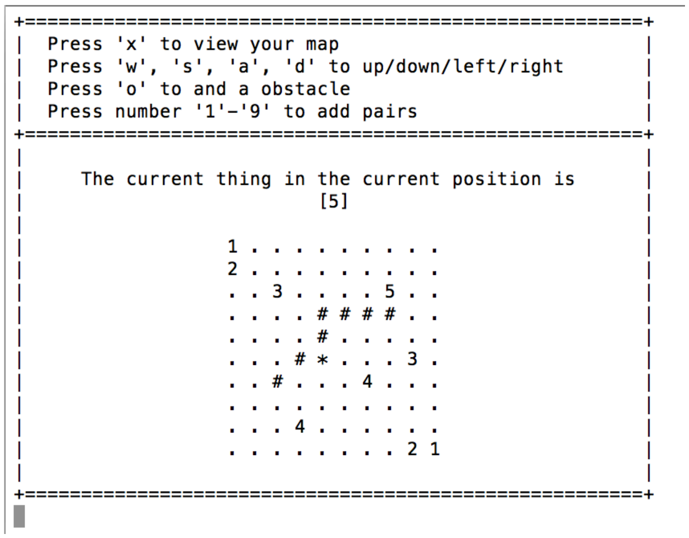
Press 'Enter' and you will see the main interface in Figure 15



```
+====================================================+
|  Press 'i' to edit the map                          |
|  Press 's' to save the map                          |
|  Press 'e' to calulate the map                      |
|  Press 'x' to get to the entrance                   |
+====================================================+
|                                                     |
|              . . . . . . . . . .                    |
|              . . . . . . . . . .                    |
|              . . . . . . . . . .                    |
|              . . . . . . . . . .                    |
|              . . . . . . . . . .                    |
|              . . . . . . . . . .                    |
|              . . . . . . . . . .                    |
|              . . . . . . . . . .                    |
|              . . . . . . . . . .                    |
|              . . . . . . . . . .                    |
|                                                     |
+====================================================+
```

Figure 15: GUI: Main Interface

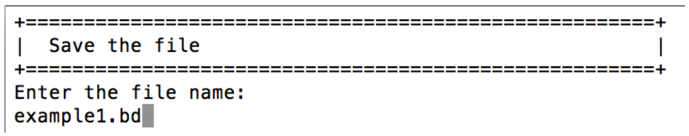
Press 'i' to enter the editor interface, and use key 'w/a/s/d', '0' '9' and 'o' to edit the map, and then press 'x' to return to the main interface (Figure 16 shows the edit interface)



```
+======================================================+
|  Press 'x' to view your map                          |
|  Press 'w', 's', 'a', 'd' to up/down/left/right      |
|  Press 'o' to and a obstacle                         |
|  Press number '1'-'9' to add pairs                   |
+======================================================+
|                                                      |
|     The current thing in the current position is     |
|                        [5]                           |
|                                                      |
|                  1 . . . . . . . . .                 |
|                  2 . . . . . . . . .                 |
|                  . . 3 . . . . 5 . .                 |
|                  . . . . # # # # . .                 |
|                  . . . . # . . . . .                 |
|                  . . . # * . . . 3 .                 |
|                  . . # . . . 4 . . .                 |
|                  . . . . . . . . . .                 |
|                  . . . 4 . . . . . .                 |
|                  . . . . . . . . 2 1                 |
|                                                      |
+======================================================+
```

Figure 16: GUI: Edit Interface

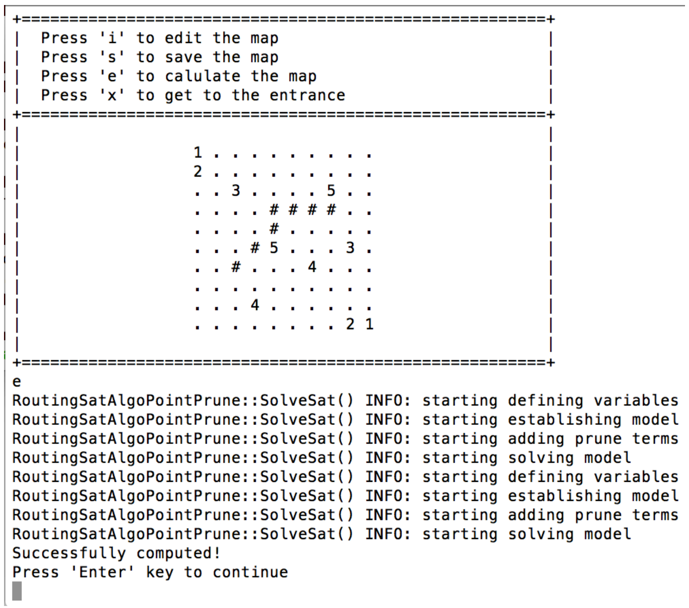
Then press 's' to save the file.



```
+======================================================+
|  Save the file                                       |
+======================================================+
Enter the file name:
example1.bd
```

Figure 17: GUI: Save Interface

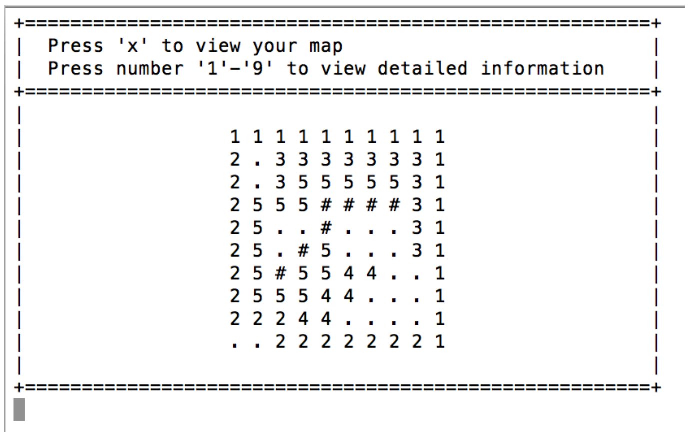
Afterwards, when returned to the main interface, press 'e' to let algorithm to calculate the answer. And you will see the interface like this in Figure 18



```
+======================================================+
|  Press 'i' to edit the map                           |
|  Press 's' to save the map                           |
|  Press 'e' to calulate the map                       |
|  Press 'x' to get to the entrance                    |
+======================================================+
|                                                      |
|                  1 . . . . . . . . .                 |
|                  2 . . . . . . . . .                 |
|                  . . 3 . . . . 5 . .                 |
|                  . . . . # # # # . .                 |
|                  . . . . # . . . . .                 |
|                  . . . # 5 . . . 3 .                 |
|                  . . # . . . 4 . . .                 |
|                  . . . . . . . . . .                 |
|                  . . . 4 . . . . . .                 |
|                  . . . . . . . . 2 1                 |
|                                                      |
+======================================================+
e
RoutingSatAlgoPointPrune::SolveSat() INFO: starting defining variables
RoutingSatAlgoPointPrune::SolveSat() INFO: starting establishing model
RoutingSatAlgoPointPrune::SolveSat() INFO: starting adding prune terms
RoutingSatAlgoPointPrune::SolveSat() INFO: starting solving model
RoutingSatAlgoPointPrune::SolveSat() INFO: starting defining variables
RoutingSatAlgoPointPrune::SolveSat() INFO: starting establishing model
RoutingSatAlgoPointPrune::SolveSat() INFO: starting adding prune terms
RoutingSatAlgoPointPrune::SolveSat() INFO: starting solving model
Successfully computed!
Press 'Enter' key to continue
```

Figure 18: GUI: Main Interface After Executed

Finally, press 'Enter' to enter the display interface, you can press '1' '5' to see more detailed infomation about each droplet (see how they step the path separately).



```
+======================================================+
|  Press 'x' to view your map                          |
|  Press number '1'-'9' to view detailed information   |
+======================================================+
|                                                      |
|                  1 1 1 1 1 1 1 1 1 1                 |
|                  2 . 3 3 3 3 3 3 3 1                 |
|                  2 . 3 5 5 5 5 5 3 1                 |
|                  2 5 5 5 # # # # 3 1                 |
|                  2 5 . . # . . . 3 1                 |
|                  2 5 . # 5 . . . 3 1                 |
|                  2 5 # 5 5 4 4 . . 1                 |
|                  2 5 5 5 4 4 . . . 1                 |
|                  2 2 2 4 4 . . . . 1                 |
|                  . . 2 2 2 2 2 2 2 1                 |
|                                                      |
+======================================================+
```

Figure 19: GUI: Display Interface

9 Conclusions

References

[1] Oliver Keszocze, Robert Wille, Krishnendu Chakrabarty, Rolf Drechsler. A General and Exact Routing Methodology for Digital Microfluidic Biochips.

[2] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In Tools and Algorithms for the Construction and Analy- sis of Systems, pages 337–340. Springer, 2008. Z3 is available at http://z3.codeplex.com/.