Team Project (V): Large-Scale Multiple-Terminal Path Finding

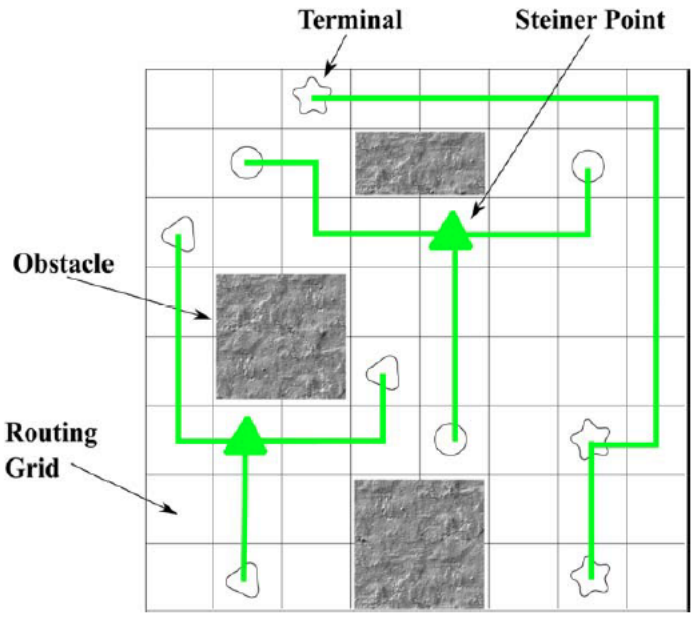
Shihong Song, Jiping Yu

June 26, 2016

Contents

1 Problem Description

Large-Scale Multiple-Terminal Path Finding



- Given: $N \times N$ routing grids, and M sets of terminals, where $N \geq 300, M \geq 10$ and each set has more than 3 terminals.
- Find: routing paths connecting each set of terminals.
- Constraints: different paths cannot cross each other, and no path can cross obstacles.
- Objective: (a) maximize the number of connected sets of terminals, and (b) when all sets of terminals can be connected,minimize the total length of all the paths.

2 Overview

Our main advantages are:

- For $N = 300, M = 10$ and each set not more than 5 terminals, the algorithm can coverage on average in 100 seconds, and for most cases ($> 99\%$) in 150 seconds.

- For larger data, i.e. larger N, M or terminals in each set, the running time will increase. However, the programming can be terminated at any time and the current best solution will be shown.
- Multiple threads are used to boost the calculation process.
- Can visualize the results into BMP format.

This document gives a method solving the problem decribed above. First, we give the overall design and system architecture of our project by giving the class design in the header file. Then, we introduce our algorithms which are used to solve this problem. Then, we give the OOP design of our project. Finally, we give the experiment results of this solving method.

3 Header Files and System Architecture

3.1 Board.h

This class is used to describe the unsolved graph given by users. It has several attributes: height, width, the points in terminal sets which need to be connected, a map describing the points given by the users in the graph, and the points of blocks which the routes cannot go through. The board has two input methods: taking a set of points as input and taking a two-dimension array sized height $\times$ width. Both the two input methods will automatically generate the terminal sets and the map of the board. The output function is used to output the board.

3.2 TerminalSet.h

`TerminalSet` is the set of the terminal points which have the index. That is to say, the points with the same index will be in the same terminal set. `TerminalSet` has a `const Board` indicating which board it belongs to, a `vector` of points which are the terminal points, a `const int id` showing its index in the graph. Users can add points to this terminal set using `AddPoint` function.

3.3 Tree.h

`Tree` is a class which is used to describe one method connecting all the points in a terminal set. `map` is a `BitMatrix` used to describe the tree

generated connecting this terminal set. `length` donates the total route-length of the tree and `computeLength()` is a function computing the length of the tree given the tree.

3.4 global.h

This file gives the definition of all the basic classes and the abbreviations used in this project. `Point` is the basic element of a graph represented by the position `x` and `y`. `Matrix` is a two-dimension `vector` which is used to store some value of the graph. `Vector` and `Pair` is inherited from STL and modification will be given if needed.

3.5 BitMatrix.h

`BitMatrix` is a bool matrix which is implemented to save memory, and also enabling fast copy and bitwise operations if needed.

3.6 Solution.h

As the name suggests, the `Solution` class is used to store the solution of the algorithm. It has a board given by the users given the original graph. `vector` trees is all the trees in the solution. `map` gives a graph of the solution which can be computed by the given trees using `computeMap()`.

3.7 Solver.h

This is the interface class for solving the problem. It only receives a `Board` and gives out a `Solution`. It will call the three strategies — solve strategy, check strategy and optimize strategy in order to accomplish its function.

3.8 SolveStrategy.h

`SolveStrategy` is the interface class for giving a board and outputting a (primal) solution. The solution will be further optimized by other methods which is not cared about by this class.

`CheckStrategy` provides the interface for checking the correctness of a solution, and `OptimizeStrategy` tries to optimize a solution.

3.9 StupidStrategy.h

This file implements three stupid strategies above: returing an empty solution, always returning legal, directly returning the original solution.

3.10 ColumnGenSolve.h

This class inherits `SolveStrategy` and implements the core (without divide-and-conquer) algorithm. The algorithm details will be discussed later.

3.11 CleverOptimize.h

This class create a new method optimizing the given solution. It inherits from the `OptimizeStrategy` and implements the optimize function. The concrete algorithm of this class will be discussed later in the algorithm part.

3.12 DACSolve.h

This class inherits `SolveStrategy` and implements the divide-and-conquer algorithm. The algorithm details will be discussed later.

3.13 test.h

This class provides a testing platform automatically calculating the running time of the test.

3.14 IntArray2bmp.h

This header file just gives a namespace that can be used to convert a two-dimension array to a picture.

3.15 GRBFactory.h

This class is the bridge between our project and Gurobi optimizer. It can configure the Gurobi enviroment and create enviroments for the project to use.

3.16 gurobi_c++.h and gurobi_c.h

These two files belongs to the Gurobi library.

4 Algorithm

4.1 Column Generation

Let's assume we have found all Steiner trees connecting each set, then the problem can easily be formulated into a integer programming problem.

Since generating all trees as well as integer programming problems are all difficult, we have to consider approximation. We can only consider the several "best" trees from each set, and both problems will be easy.

We only need to define "good" trees and find them.

First we construct the current best solution.

For the sets in the solution, we regard other sets' trees in the solution as obstacles and try to reroute the current set.

For each set, we run a new optimizing problem without the current set, and then regard other sets' trees in the solution to the new problem as obstacles and try to route the current set.

We now only need a fast algorithm to route a set on a board with obstacles.

The classic shortest Steiner tree algorithm needs $O(3^t n^2 + 2^t n^2 \log n)$ time, where t is the number of terminals of the current set. It is OK with $t = 3, n = 300$ or like, but the algorithm is limited because it is unacceptable for any little larger t. In addition, it is no use to make the algorithm so precise in this step (since it is already a large approximation above).

Our algorithm is based on Dijkstra's shortest path algorithm. We perform Dijkstra's algorithm from each terminal and enumerate the center. After connecting, the tree is not optimal usually (unless $t \leq 3$). We enumerate each edge and try to find another edge instead, where Dijkstra's algorithm is used again. We will stop optimize the tree when no possible edge can be improved.

In addition, the Dijkstra results do not affect each other, so the algorithm is performed by multiple threads. Because our computing resource is limited, we only use two threads to boost the algorithm.

When the solution is not improved after an iteration, the algorithm will terminate (though it is possible to improve if we do not terminate).

4.2 Divide and Conquer

Only using the algorithm above can get good results, but the divide-and-conquer strategy can improve the timing performance (though maybe decrease the solution quality a little).

The strategy is simple. Divide the board into 4 parts as equal as possible. Solve the problem in each board. Assuming the points on trees of solution to the subproblems are all "fixed" terminals on the original board and do the final solve process.

Thus, the solution space is more limited, and the running time will be improved. It is possible that the total number of the used grids may increase, but the number of connected sets will almost always improve.

4.3 CleverOptimize

This algorithm is used to optimize some solutions which are not good enough. Those which can be optimized will have the following attributes: the leaf of the tree can have a shorter route to the other routes of the tree. The algorithm implements as follows:

Find the terminal set of the tree, go through every terminal of the tree and check each terminal point:

Find the nearest point on the route which is connected to this point and has more than 3 degrees. Erase all the route between the terminal point and this point and again, find the shortest path from the terminal point to the left tree. This will guarantee the optimized route have the shorter length than the previous one. The process can be implemented by using simple bfs and backtrace. Note that if the found route which needs optimizing cross another terminal point, this route should not be optimized in case it makes another terminal point not connected to the route. If there is no point on the route having more than 3 degrees, we just put back the erased points to make sure the solution doesn't become worse. Implementing this algorithm has lots of incidents to consider about: the blocks which will be taken into account in the bfs process, the bfs process will also take the other terminal trees into account and this should not interfere the current route, etc.

5 OOP Design

There are mainly four types of classes.

Utility classes: `Vector<T>`, `Matrix<T>`, `BitMatrix`, etc. These classes are used for coding convenience.

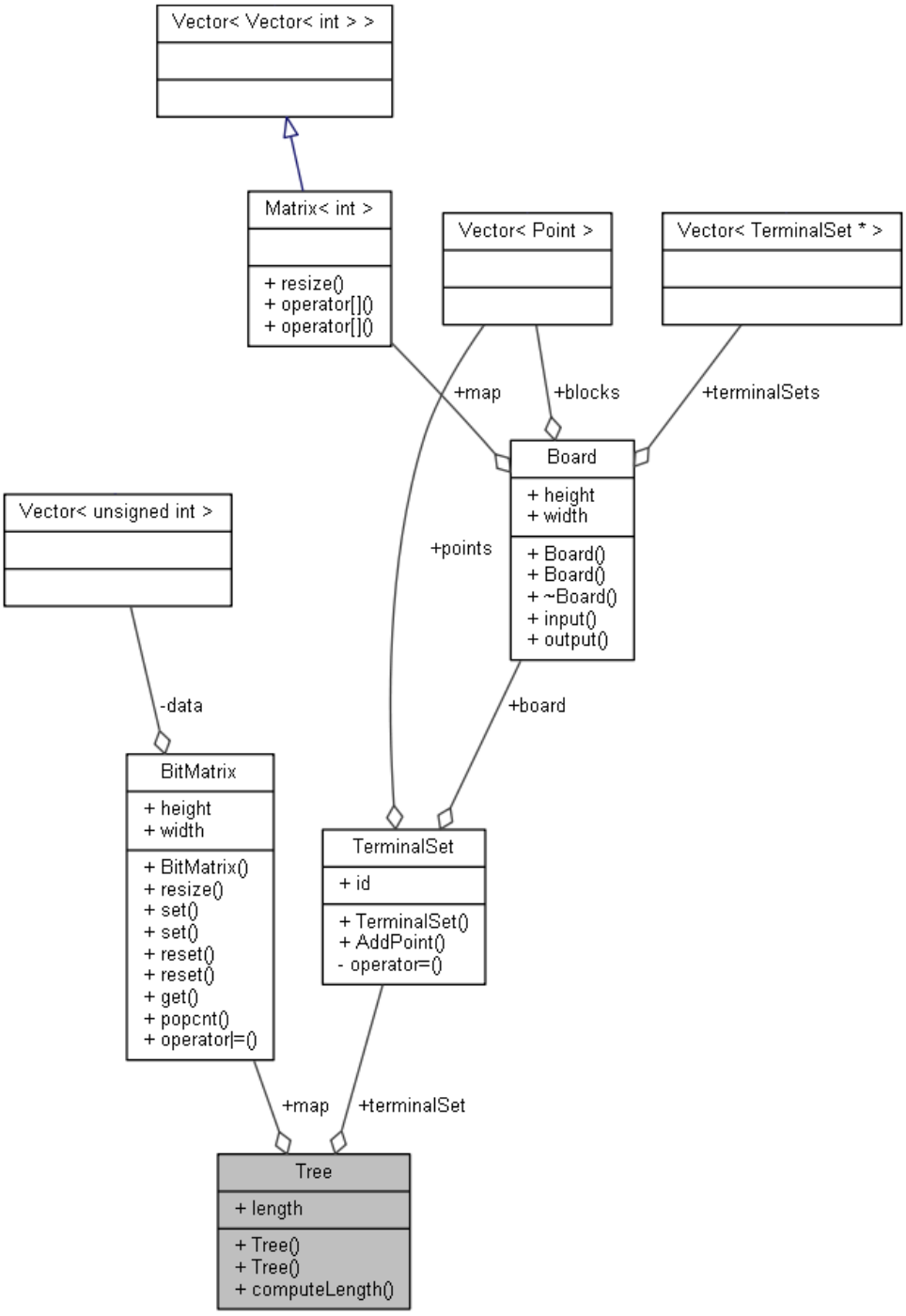
Basic classes: `Board`, `TerminalSet`, `Tree`, etc. These classes provide basic data storage and functions for the problem description.

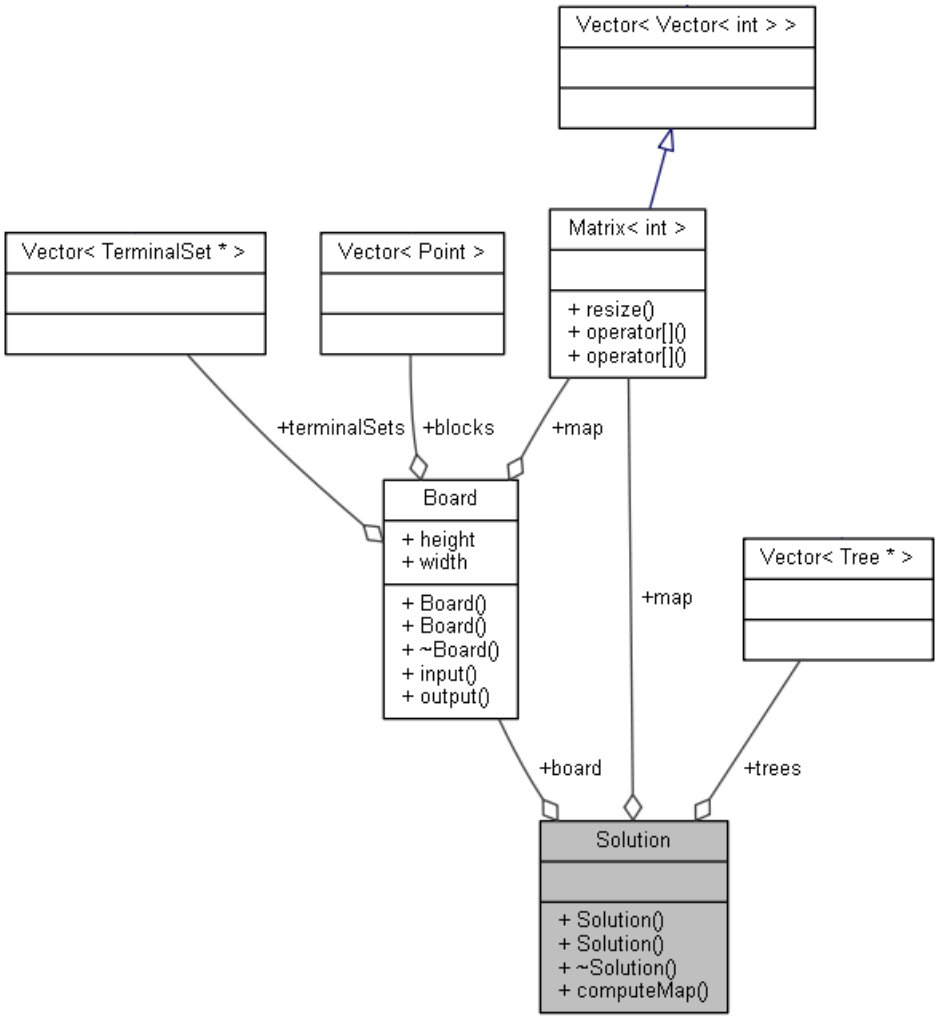
Interface classes: `Solver`, `SolveStrategy`, `CheckStrategy` and `OptimizeStrategy`. These classes provide interfaces for solving the given problem. Certainly, polymorphism is used here.

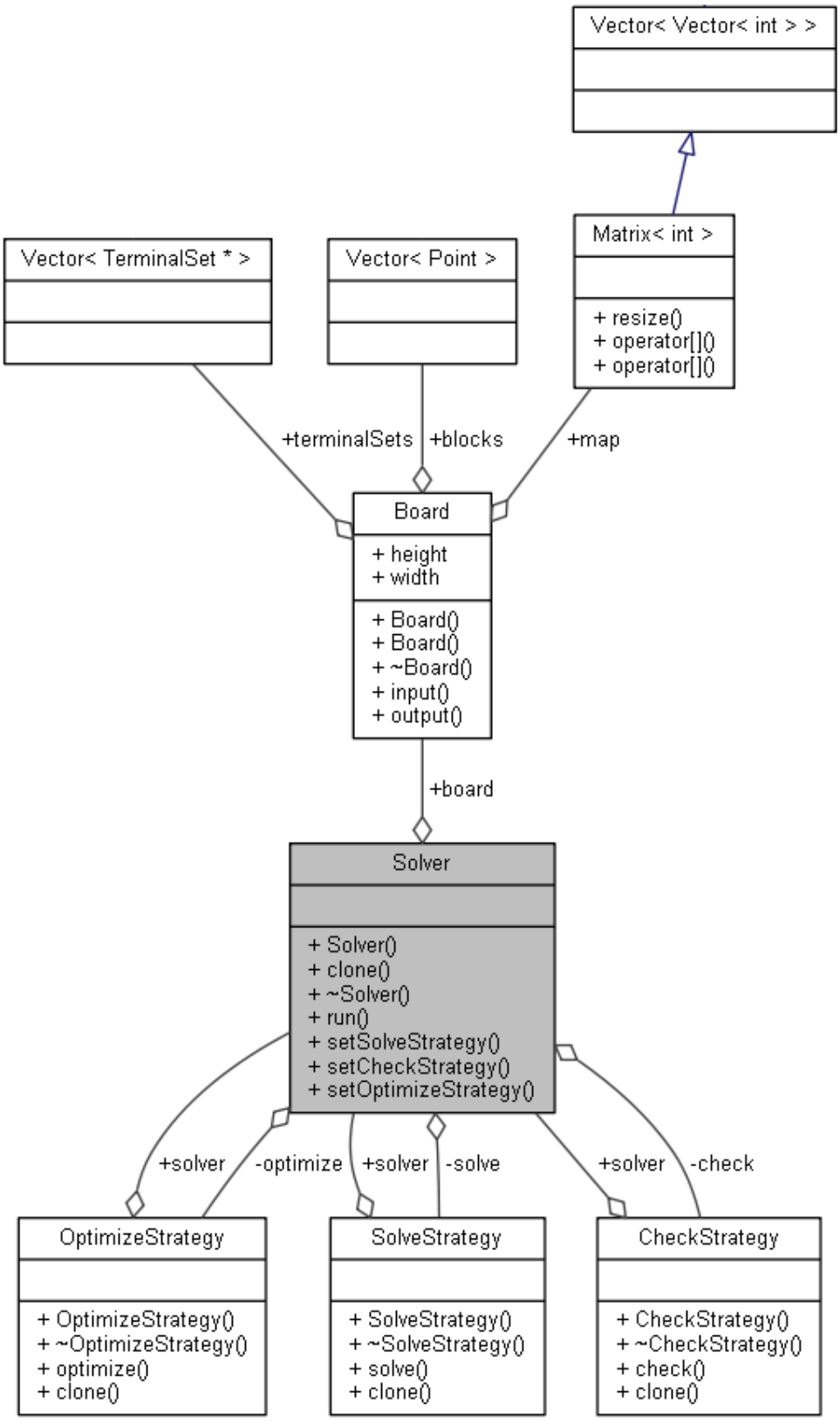
Implementation classes: `ColumnGenSolve`, `DACSolve`, `CleverOptimize`, etc. These classes implement the interfaces above.

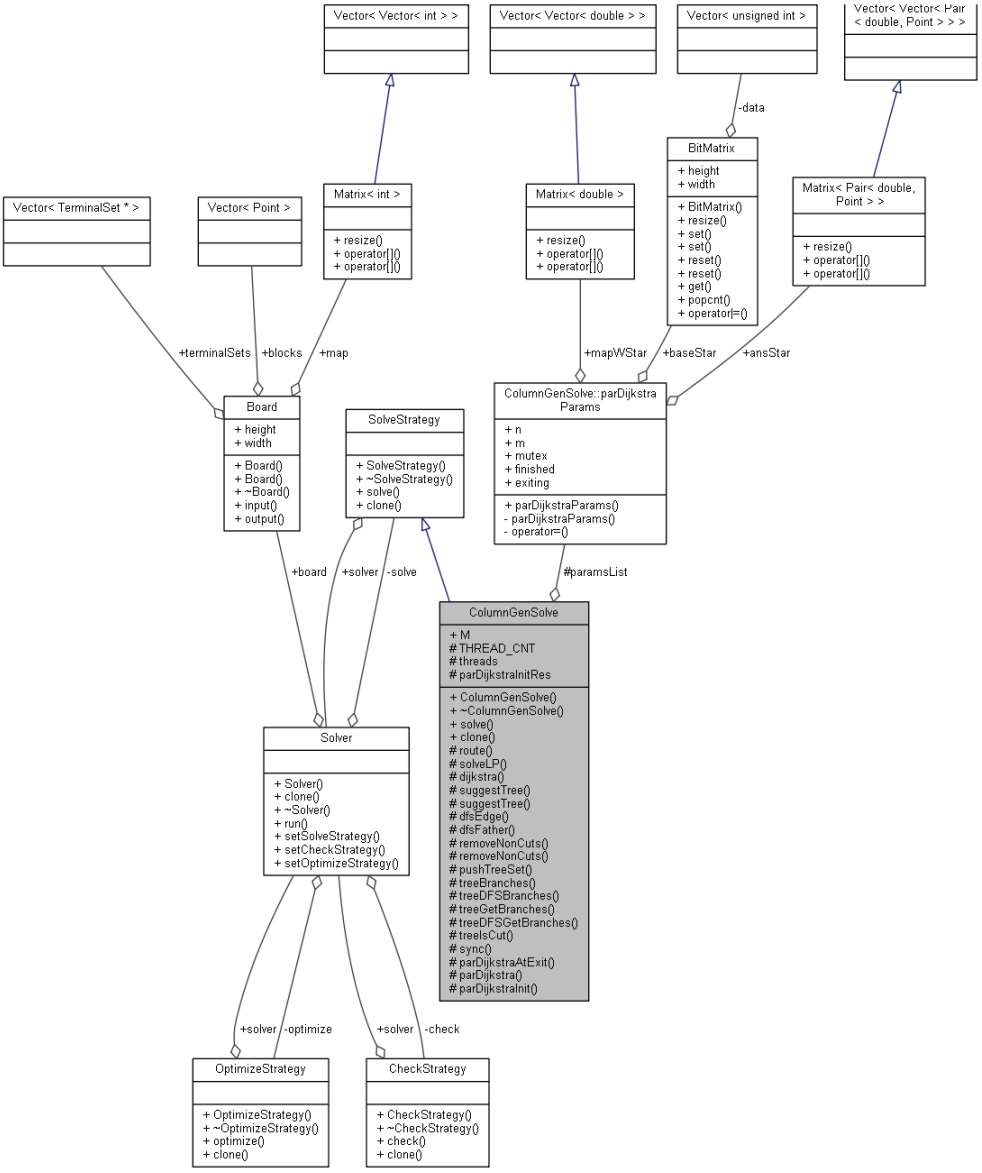
It is noticeable that strategy design pattern is used here, for easily alternating the solving strategy.

Some useful graphs describing the relationships between the classes are shown here.


Vector< Vector< int > >
Matrix< int >
+ resize()
+ operator[]()
+ operator[]()
Vector< Point >
Vector< TerminalSet * >
+map
+blocks
+terminalSets
Board
+ height
+ width
+ Board()
+ Board()
+ ~Board()
+ input()
+ output()
Vector< unsigned int >
+points
-data
+board
BitMatrix
+ height
+ width
+ BitMatrix()
+ resize()
+ set()
+ set()
+ reset()
+ reset()
+ get()
+ popcnt()
+ operator|=()
TerminalSet
+ id
+ TerminalSet()
+ AddPoint()
- operator=()
+map
+terminalSet
Tree
+ length
+ Tree()
+ Tree()
+ computeLength()


Vector< Vector< int > >
Matrix< int >
+ resize()
+ operator[]()
+ operator[]()
Vector< TerminalSet * >
Vector< Point >
+terminalSets
+blocks
+map
Board
+ height
+ width
+ Board()
+ Board()
+ ~Board()
+ input()
+ output()
+map
Vector< Tree * >
+board
+trees
Solution
+ Solution()
+ Solution()
+ ~Solution()
+ computeMap()


Vector< Vector< int > >
Matrix< int >
+ resize()
+ operator[]()
+ operator[]()
Vector< TerminalSet * >
Vector< Point >
+terminalSets
+blocks
+map
Board
+ height
+ width
+ Board()
+ Board()
+ ~Board()
+ input()
+ output()
+board
Solver
+ Solver()
+ clone()
+ ~Solver()
+ run()
+ setSolveStrategy()
+ setCheckStrategy()
+ setOptimizeStrategy()
+solver
-optimize
+solver
-solve
+solver
-check
OptimizeStrategy
+ OptimizeStrategy()
+ ~OptimizeStrategy()
+ optimize()
+ clone()
SolveStrategy
+ SolveStrategy()
+ ~SolveStrategy()
+ solve()
+ clone()
CheckStrategy
+ CheckStrategy()
+ ~CheckStrategy()
+ check()
+ clone()



6 Experiment Results

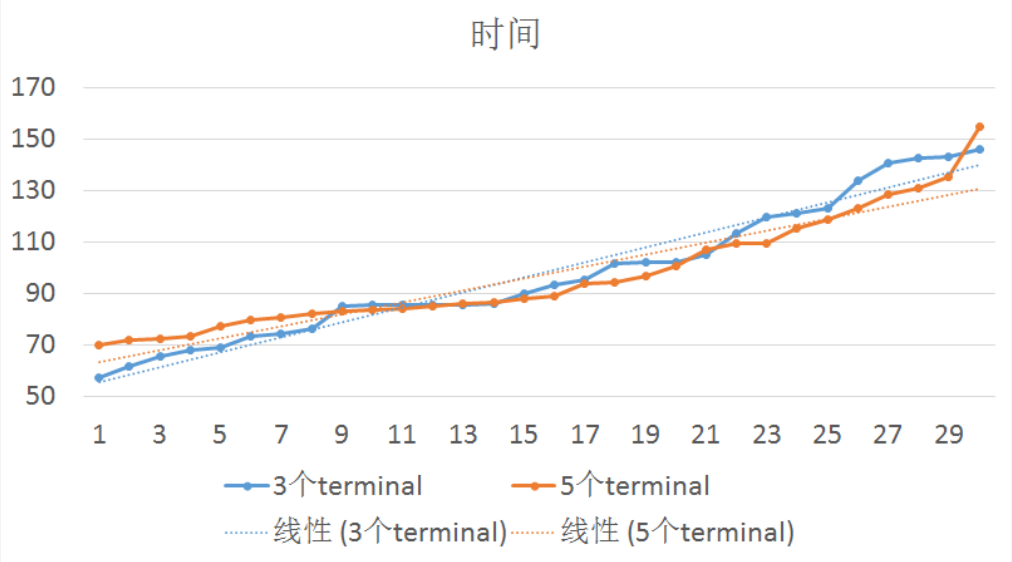
Here is a table of the 10 testcases:

id	0	1	2	3	4*
Size	300	300	300	400	300
Set	10	10	10	10	100
Terminal	3	3	3	3	3
Block($\times 10^4$)	0	1	2	1	1
Time	72.562s	95.673s	135.041s	323.812s	N/A*
Connected Sets	9	9	9	10	24
Length	3490	3494	4060	6693	6693

id	5	6	7	8	9
Size	300	300	300	350	350
Set	30	10	10	10	10
Terminal	3	5	10	3	5
Block($*10^4$)	1	1	1	1	1
Time	25m8.331s	90.156s	736.371s	159.130s	258.372s
Connected Sets	19	5	4	9	7
Length	6783	2817	3735	4481	5387

*: The result is obtained at 27m24.842s. We did not wait until its termination.

We also ran 60 tests for 300 × 300 boards, 10 sets, each with 3 or 5 terminals, and from 0 to 10085 blocks (randomly generated) and record the running time. The graph is shown:



The average running time of 3 and 5 terminals are 97.642 and 97.001 seconds. It is observed that, when the number of terminals of each set is

not so large (≤ 5), the running time does not explicitly depends on it.