2017年3月26日

Solving Graph Coloring Problem by Genetic Algorithm

Using Pure C Language

Kui Chen
UNIVERSITY OF TSUKUBA JAPAN

Content

Abstract

Graph coloring problem (GCP) is a famous combinatorial optimization problem (COP). It states that given an undirect graph with n nodes and m edges, each node is assigned one of k colors so that nodes connected with an edge have different colors. In this document, we describe how to solve graph 3-coloring problems with genetic algorithm (GA).

1. Overview of Project

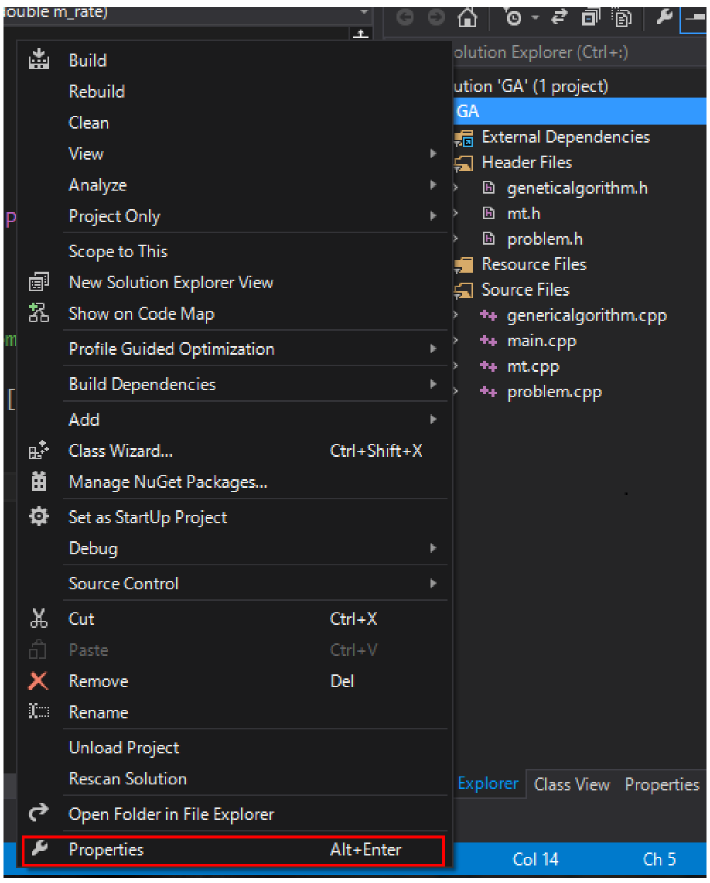
This project is built in pure C language with Visual Studio 2013 Ultimate Edition. The main five files in this project are listed as below:

File name	Description
`problem.h`	Definition of macro and functions which build random graph.
`problem.cpp`	The implementation of functions defined in problem.h head file.
`geneticalgorithm.h`	Definition of macro and functions which establish genetic algorithm.
`geneticalgorithm.cpp`	The implementation of functions defined in geneticalgorithm.cpp
`main.cpp`	Controlling the whole procedure of solving graph 3-coloring problems.

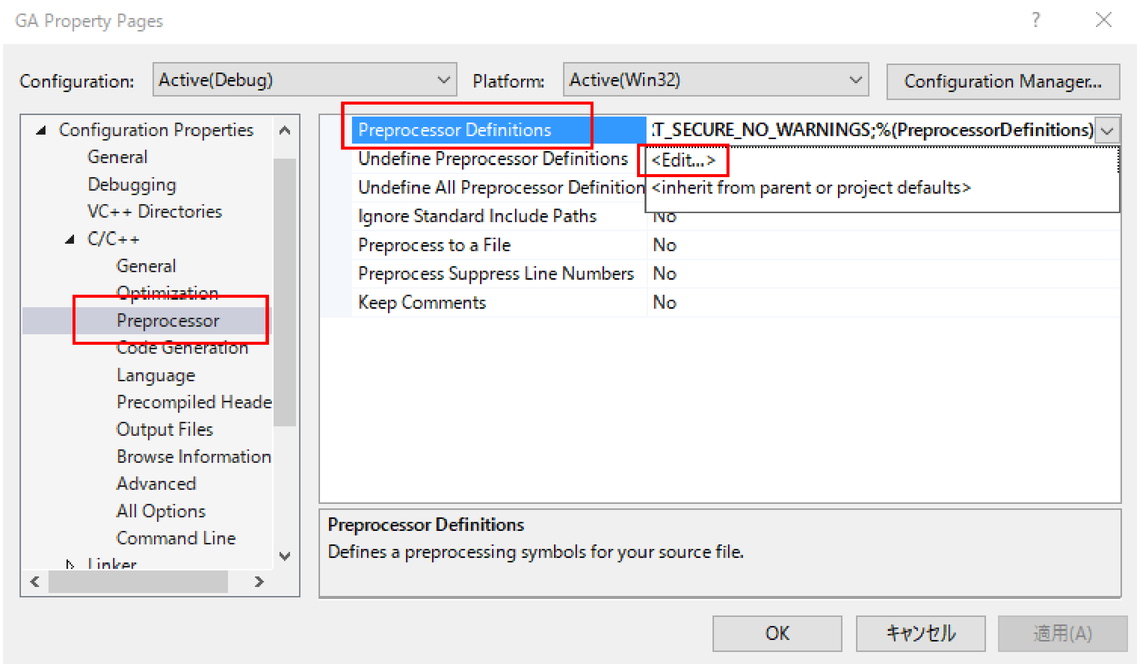
We will describe the most functions of each file in next parts.

Note: <u>if you want to test this program in Visual Studio environment, you should do some settings before running it. The procedure is shown as below steps:</u>

a. Right click the project name and select properties.



b. 【property pages】⇒【configuration properties】⇒【C/C++】⇒【Preprocessor】⇒【Preprocessor Definitions】⇒【Edit...】

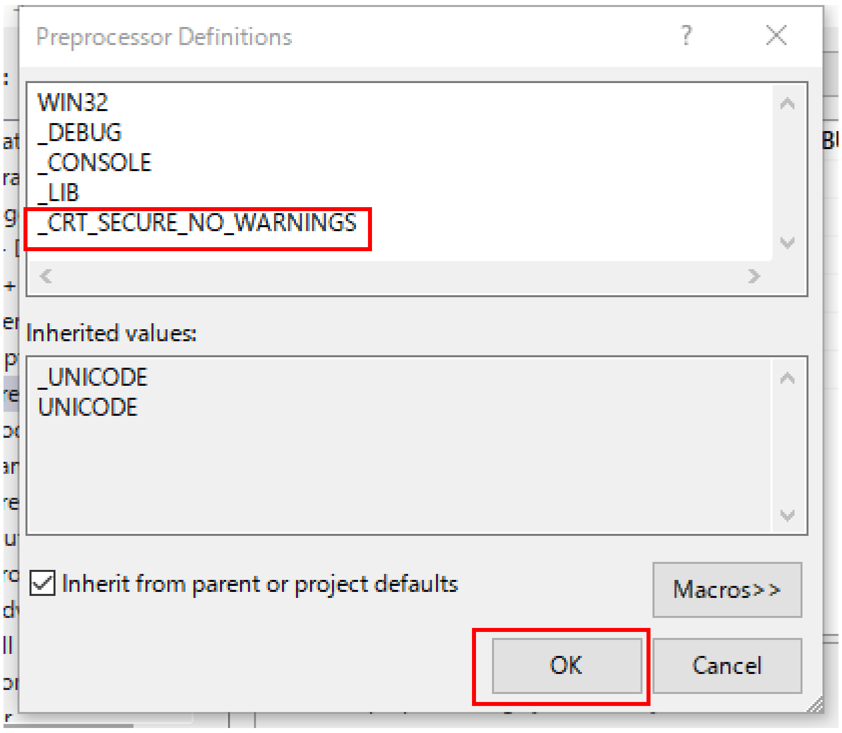


Configuration: Active(Debug)   Platform: Active(Win32)

Preprocessor Definitions	;T_SECURE_NO_WARNINGS;%(PreprocessorDefinitions)
Undefine Preprocessor Definitions	<Edit...>
Undefine All Preprocessor Definition	<inherit from parent or project defaults>
Ignore Standard Include Paths	No
Preprocess to a File	No
Preprocess Suppress Line Numbers	No
Keep Comments	No

Preprocessor Definitions
Defines a preprocessing symbols for your source file.

c. Add "`_CRT_SECURE_NO_WARNINGS`" in Preprocessor Definitions, then click OK button to save settings.


Preprocessor Definitions

```
WIN32
_DEBUG
_CONSOLE
_LIB
_CRT_SECURE_NO_WARNINGS
```

Inherited values:

```
_UNICODE
UNICODE
```

☑ Inherit from parent or project defaults

The reason of setting like above is that "`printf`" and some other functions in ANSI C are considered unsafe in Visual Studio 2013. Adding "`_CRT_SECURE_NO_WARNINGS`" can avoid error information when compiling the project.

The structure of this document is as follow: first, I will give the algorithm by which we generate random graph. After that, the GA procedure are described step by step. Then, how to use my code is given in section 4. Finally, the result and future work are shown in section 5 and 6.

2. Build Problem

2.1 Establishment of Problem

Firstly, we must build random graph, which is the base of our project. The method of generating a n nodes and m edges random graph is shown as below:

a. Dividing nodes into 3 groups, each with $n/3$ nodes

b. Creating edges randomly between nodes until the total number of edges is m

This method is similar with Minton's method which is given in [1].

Second, let red be 0, blue be 1 and green be 2, a solution vector $\boldsymbol{x}$ is define as:

$$\boldsymbol{x} = (x_1, x_2, \dots, x_n)^T \text{ where } x_i \in (0, 1, 2)$$

Given a random graph with n nodes and m edges and a solution vector $\boldsymbol{x}$, the conflict function *conflict*($\boldsymbol{x}$) is used to calculate the conflict number of this solution for the graph.

The target of this project is to find a solution vector whose conflict number is 0, that is *conflict*($\boldsymbol{x}$) = 0. This means, by coloring the graph with this solution, there is no such two connected nodes in the graph that have the same color.

In the source code, the object function $f(\boldsymbol{x})$ is given as:

$$f(\boldsymbol{x}) = 1 - \frac{conflict(\boldsymbol{x})}{m}$$

If conflict number of a solution is 0, then the fitness of this solution is 1.

The difficulty of a given random graph can be measured by constraint density d, which is defined as:

$$d = \frac{m}{n}$$

According to Hogg's research [2], the most difficult problem occurs when d = 2.5. This can be shown at section 5.

2.2 Description of Source Code

In the project, all code associated with graph is in two files, `problem.h` and `problem.cpp`. A random graph is defined as a 2-d array and be generated by function:

```
void generate_random_graph(char(*graph)[NODE_NUMBER], float
d);
```

The method generating random graph is identical with the method introduced in 2.1.

The fitness of a solution vector is calculated by

```
double fitness(char const (*graph)[NODE_NUMBER], char const
*solution);
```

and we save the conflict information to a structure:

```
typedef struct graph_conflict_list {
      int conflict_nodes[NODE_NUMBER];
      int conflict_numbers[NODE_NUMBER];
      int len;
      int max_conflict_node;
      int max_conflict;
}Conflict_Infor;
```

For example, given a 4 nodes graph, you may get the following conflict information: node 1 conflicts with 3 other nodes, node 2 conflicts with 2 other nodes, node 3 and node 4 have no conflict with any other nodes. Then the `conflict_nodes` array is `{1, 2, 3, 4}` and the `conflict_numbers` array is `{3, 2, 0, 0}`. The component `len` identifies the number of nodes whose conflict number is not 0, here it is 2. The `max_conflict_node` is 1 and the `max_conflict` is 3.

Structure can be calculated by function:

```
int solution_conflict(char const (*graph)[NODE_NUMBER], char
const *solution, Conflict_Infor *conflict_infor);
```

and a graph can be saved by:

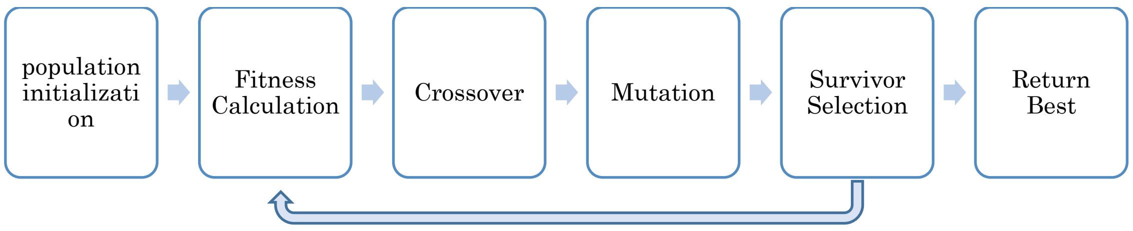
```
int save_graph(char const (*graph)[NODE_NUMBER], char const
*filename, char const *extension);
```

you can find the detailed implementation in `problem.cpp`.

3. Genetic Algorithm

3.1 Procedure of GA

Genetic Algorithm (GA) is a search-based optimization technique based on the principles of Genetics and Natural Selection. It is simply and used to find optimal or sub-optimal solutions to difficult problems which otherwise would take a lifetime to solve. It is frequently applied to solve optimization problems, in research, and in machine learning. The basic framework of GA is shown as below:



In the project, a chromosome is defined as a structure which has two elements: a 1-d array, which identifies a candidate solution $\boldsymbol{x}$, and a `double` value, which is the fitness value of candidate solution.

```
typedef struct Chromosome {
      char solution[NODE_NUMBER];
      double fitnessValue;
} Chromosome;
```

In next sections, we will cover each important step of GA and explain most functions in file `geneticalgorithm.h` and `geneticalgorithm.cpp`.

3.2 Selection

There are many select method in GA. Here, in this project, we have implemented two of them: roulette wheel selection and tournament selection.

The roulette wheel selection is a selection method in which fitness value is used to associate a probability of selection with each individual chromosome. If f_i is the fitness of individual i in the population, its probability of being selected is

$$p_i = \frac{f_i}{\sum_{i=1}^{N} f_i}$$

Where the N is the number of individuals in the population. The larger the probability, the higher chance of this individual can be selected. The roulette wheel selection is implemented in:

```
unsigned    int    roulette_selection(Chromosome    const
*chromo_list, double sum_fitness);
```

On the other hand, the tournament selection, takes k candidates from chromosome pool and returns the best one of them. It is defined in:

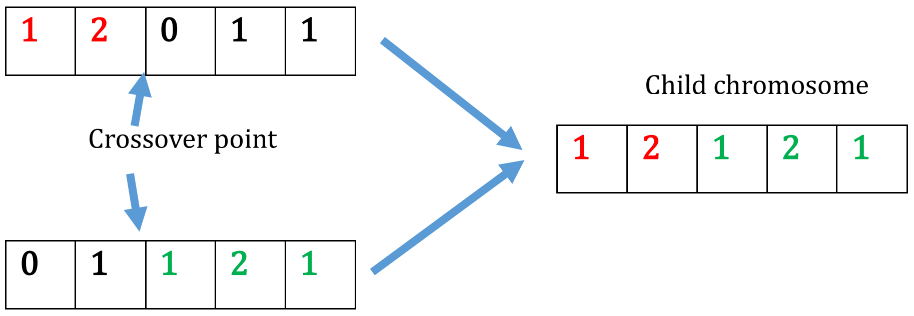
```
unsigned   int   tournament_selection(Chromosome   const
*chromo_list);
```

and you can add other selection methods to the project.

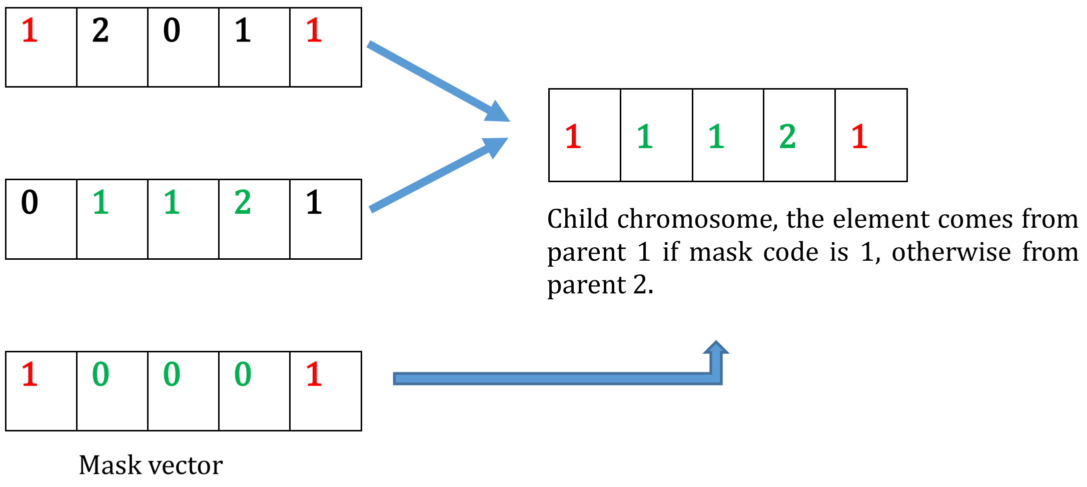
3.3 Crossover

In GA framework, a new child chromosome is generated by crossover two parent chromosomes. In this project, two crossover methods are implemented: point crossover and mask crossover.

After choose a crossover point randomly, the point crossover combines part of two parent chromosomes and generates a new child. It is shown as below:



The mask crossover generates a 0-1random vector as a mask to determine where each element of child chromosome comes from. It is shown as below:


1 2 0 1 1
0 1 1 2 1
1 1 1 2 1
Child chromosome, the element comes from parent 1 if mask code is 1, otherwise from parent 2.
1 0 0 0 1
Mask vector

We implemented crossover in function:

```
void    crossover(Chromosome    const    *parent_chromo_list,
Chromosome *children_chromo_list);
```

and you can add your crossover methods in the `switch` statement.

3.4 Mutation

Mutation is also an important step in GA. It mutates an element of a chromosome by a probability called mutate rate. After mutation, part of elements has converted into other values. The larger the mutate rate, the higher chance the element will be changed. Mutation is implemented as function:

```
void mutation(Chromosome *chromo, double m_rate);
```

3.5 Other Useful Functions

I have designed a function called "`elapsed_times`" to record the time from beginning of GA to the end of it. This function returns a string "`used_time`" as how many seconds the GA takes:

```
void elapsed_times(time_t const *start_time, time_t const
*end_time, char *used_time);
```

The result of GA is recorded in a structure:

```
typedef struct Result {
      int success;
      int loop_times;
      double eval_times;
      double gbest_list[MAX_LOOP];
      char solution[NODE_NUMBER];
      char start_time[50];
      char end_time[50];
      char s_elapsed_times[100];
} Result;
```

In this structure, `success` identifies finding optimal value (1) or not (0); `loop_times` is the generation number when GA finishes; `eval_times` is how many times the object function is calculated; `gbest_list` records current global best of each generation; `solution` is the solution vector finally found; `start_time`, `end_time` and `s_elapsed_times` are start time, end time and cost time of GA respectively.

Finally, using below function to start GA:

```
Result*genetic_algorithm(char const (*graph)[NODE_NUMBER]);
```

and saving the result as a csv or txt file by:

```
int save_result(Result const*result, char const *file_name);
```

4. How to Use My Code

My code is well designed so that you can use it just by setting a few values of parameters and need not modify any source code. The part of settings is shown below:

```
problem.h file:

#define NODE_NUMBER     90
```

This macro is used to set the size of graph.

```
geneticalgorithm.h file:

#define POP_SIZE  200
#define MAX_LOOP  10000
......
#define MUTATE_RATE     0.014
......
#define CROSS_METHOD    2
#define SELECT_METHOD   2
#define K_CANDIDATE     2
#define USE_HYBRID      0
......
#define PRINT_DETAIL    0
```

All GA settings are in `geneticalgorithm.h` file. `POP_SIZE`, `MAX_LOOP` and `MUTATE_RATE` are population size, max generations and mutation rate respectively. `CROSS_METHOD` can be 1 or 2, identifies point crossover and mask crossover. `SELECT_METHOD` can be 1 or 2, identifies roulette wheel selection or tournament selection. If tournament selection is used, the `K_CANDIDATE` is the number of candidate chromosomes. If you do not need hybrid algorithm to improve the performance, set `USE_HYBRID` to 0, otherwise 1. Finally, if you need the details of each step, set `PRINT_DETAIL` to 1.

```
main.cpp

#define MAX_RUN   30
......
```

For the sake of fairness, 30 independent runs are observed for each constraint density *d* (see section 2.1) because of the stochastic nature of GA. You can set run times by `MAX_RUN`.

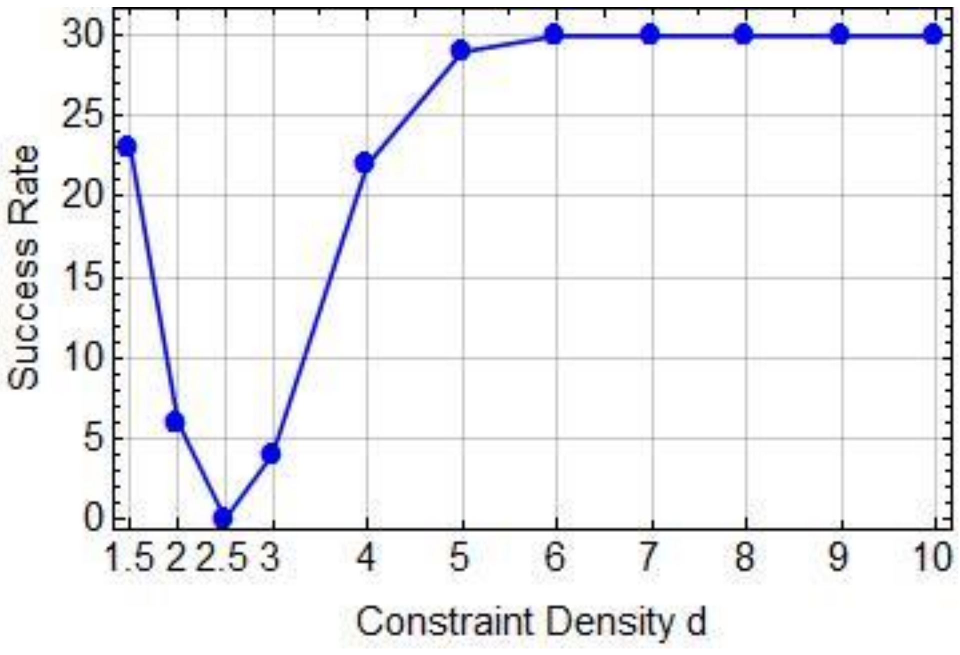
5. Result

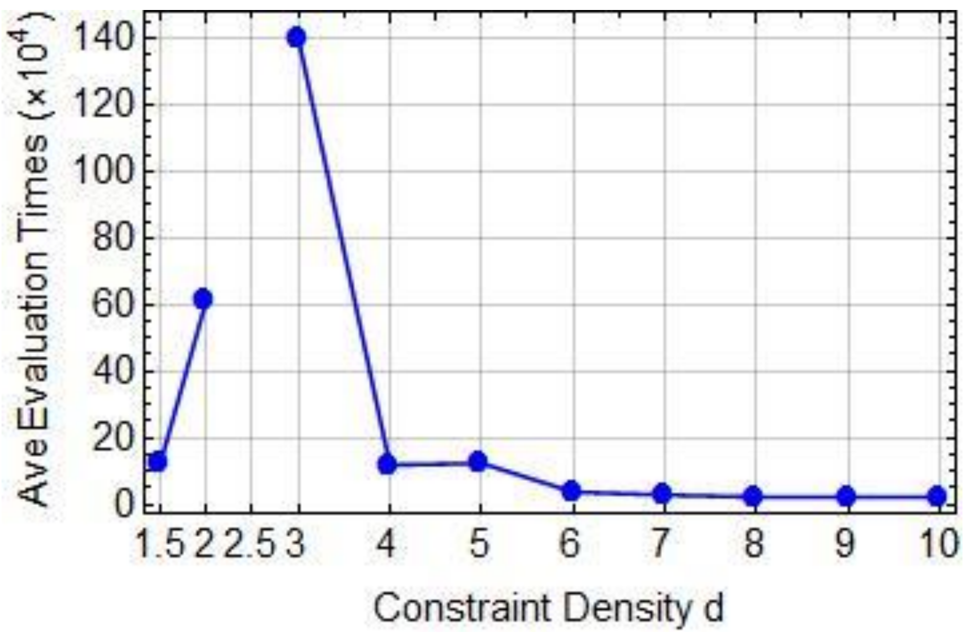
By using these settings, we get the results below:

POP_SIZE	200
MAX_LOOP	10000
Selection	Tournament selection
Crossover	Mask crossover
Mutation rate	0.014
Candidate number	2
MAX_RUN	30
Constraint density *d*	1.5~10

Node Number of Graph	90

d	Success rate	Average evaluation times
1.5	23	1.28E+05
2	6	6.15E+05
2.5	0	----
3	4	1.40E+06
4	22	1.19E+05
5	29	1.24E+05
6	30	3.79E+04
7	30	2.90E+04
8	30	2.25E+04
9	30	2.23E+04
10	30	2.33E+04





As it is shown in these results, we find that $d = 2.5$ is the most difficult problem and cannot get the optimal solution. This is the proof of Hogg's research [2].

You may want to hybrid GA with some other algorithms to improve its performance. In the project, I have implemented two algorithms: assessment strategy and hill climbing at `geneticalgorithm.h` file:

```
int  assessment_strategy(char  const  (*graph)[NODE_NUMBER],
Chromosome *chromo);

int    hill_climbing(char    const    (*graph)[NODE_NUMBER],
Chromosome *current_chromo);
```

and you can use hybrid algorithm by setting:

```
#define USE_HYBRID     1
```

6. Future Work

The future work is focused on updating my C code to C++ so that GA can be more flexible and applied to solve many other problems.

Reference

[1] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. *Minimizing conflicts: A heuristic repair method for constraint-satisfaction and scheduling problems*. Artificial Intelligence, 58:161-205, 1992.

[2] Tad Hogg and Colin Williams. *The hardest constraint problems, a double phase*

transition. Artificial Intelligence, 69(1-2):359-377, 1994

Thank You for Your Reading! If You Have Any Question or Advice, Please Contact with Me: zbchenkui@outlook.com. ***I Am Happy to Hear Feedback from You!***