



Genetic Algorithm and Tabu Search for K-Graph Coloring Problem

By Jinmei GAO & Abdelhamid Sta

Université d'Evry - paris saclay, France

*M2 Genomics Informatics and Mathematics
for Health and Environment*

1. Formal definition of the problem	P3
1.1 Definition of K-Graph Coloring problem	P3
1.2 Fitness function explanation and improvement	P4
2. Random Generation of an instance of the problem	P6
3. Explanation of problem coding and the fitness function	P7
3.1 Genetic Algorithm	P7
3.1.1 Definition of Genetic Algorithm	P8
3.1.2 Six steps for solving this problem by using Genetic Algorithm ..	P8
3.2 Tabu Search	P13
3.2.1 Definition of Tabu Search	P13
3.2.2 Setting and running TABU algorithm	P14
4. Experimental evaluation	P19
4.1 Comparison in two Fitness functions	P19
4.1.1 Two Fitness functions comparison by using Genetic Algorithm	P19
4.1.2 Two Fitness functions comparison by using Tabu Search	P19
4.2 Comparison in small, medium and large graphs	P24
4.2.1 Comparison in different graphs by using Genetic Algorithm ..	P24
4.2.1 Comparison in different graphs by using Tabu Search	P25
4.3 Comparison in Genetic Algorithm and Tabu Search	P27
5. Conclusion	P27
6. Program in appendix	P27
7 . References	P27

1. Formal definition of the problem

1.1 Definition of K-Graph Coloring problem

The k -graph coloring problem ($K - GCP$) consists in assigning a color (from a number in $\{1, \dots, k\}$) to each vertex of a graph $G = (V, E)$ where V is the number of vertices and E the number of edges and K a positive integers. The followed K -coloring graph will be the assignment of K distinct colors to each vertex $v \in V$ in the graph, so that no two adjacent vertices (linked by an edge $e \in E$) are given the same color, $\forall \{V_i, V_j\} \in E, c(i) \neq c(j)$. The problem is to determine if a graph that can be assigned a proper K -coloring or less exists. If such a color exists, G is called K -colorable.

We can also extend K -GCP to other graph problems. In particular, there is the graph coloring problem (G -COL). This is an optimization problem aimed at finding the smallest K such that G is K -colorable. The smallest K will therefore be called the chromatic number of G . If K -GCP is solvable, we can also solve G -COL by following the iterative approach: 1) Finding a valid K color for a graph G (solving K -GCP) 2) Iteratively decrease K , with $K = K - 1$ and rerun K -GCP, until no conflict free K coloring graph can be found. The value of K will then be considered as the smallest possible for the graph G , and will represent its chromatic number.

The $K - GCP$ is computationally hard to solve, and it is NP-complete to identify if a given graph admits or not, a K -coloring for a given K .

This completeness is notably verifying following the given properties :

- 1) It is possible to verify a solution efficiently (in polynomial time); the problems verifying this property is firstly denoted as NP
- 2) All the problems of the class NP can be reduced to this one via a polynomial reduction; this means that the problem is at least as difficult as all the other problems of the class NP. The problem can then be denoted as NP-Complete

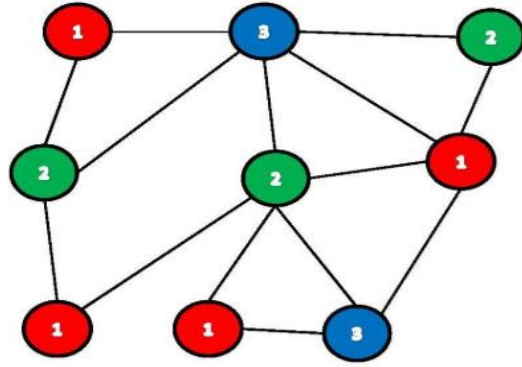


Fig 0 : Randomly generated k-colorable graph given k=3

Coloring problems are at the heart of numerous applications including, for instance, scheduling, register allocation in compilers and frequency assignment in mobile networks. In the following paper, we will consider the k-graph coloring problem (K - GCP) using two Metaheuristic approaches, the Genetic algorithm and tabu search algorithm.

1.2 Fitness function explanation and improvement

In the case of the K-GCP , an optimal fitness function will overall allow us to know how close a given design solution is to achieving the set aims. . The function will be computed as follow :

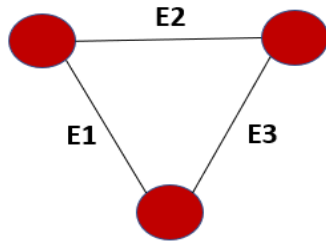
$$\text{Fitness score}(1) = \sum s(i,j) \quad \text{for } \forall \{V_i, V_j\} \in E$$

$$\text{where } s = 1 \quad \text{if } \{V_i, V_j\} \in E \text{ and } c(i) = c(j)$$

$$s = 0 \quad \text{if } \{V_i, V_j\} \in E \text{ and } c(i) \neq c(j)$$

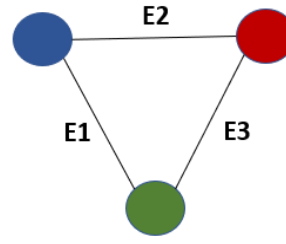
The fitness function will do the sum of all the edges, where the edge possessing two same colored vertices will be of value 1 , while no same colored vertices will be of value 0 . A not corrected K coloring will have its fitness function > 0 (fig1.A) while a correct K will be of fitness = 0 , and therefore correspond to a valid K - coloring graph (fig 1.B) . The goal will

be to consider the solutions who tend to minimize this function until it reaches the value of zero .



$$\text{Fitness: } \sum E(i,j) = (E1) 1 + (E2) 1 + (E3) 1 = 3$$

Fig 1.A non valid graph



$$\text{Fitness: } \sum E(i,j) = (E1) 0 + (E2) 0 + (E3) 0 = 0$$

Fig 1.B K = 3 colorable graph

However, that first fitness function does not allow us to discriminate graphs having the same number of conflicts and therefore constitute a limitation . Indeed, the number of conflicts shouldn't constitute the unique indication to evaluate the graph. As depicted below, in graph 2 , a K-GCP with K = 3 , while both have only one conflict V5 - V4 and V5-V3 , the solvability of this graph does not require the same number of steps . The Fig2.B requires one step from switching V5 : RED \Rightarrow V5 : Green , but the Fig2.A requires at least two steps V3 : Blue \Rightarrow V3 : Green, then V2 : Green \Rightarrow V2 : Blue .

The new fitness should allow us to take into account this new complexity, bringing- by the degree of vertices. A vertex with a higher degree is more complex to solve since we are more likely to perturb the rest of the configuration . To consider the degree of vertices (deg(V)) we developed the following function

$$\text{Fitness score}(2) = \sum s(i,j) * (\text{degree}(Vi) + \text{degree}(Vj)) \text{ for } \forall \{Vi,Vj\} \in E$$

$$\text{where } s = 1 \text{ if } \{Vi,Vj\} \in E \text{ and } c(i) = c(j)$$

$$s = 0 \text{ if } \{Vi,Vj\} \in E \text{ and } c(i) \neq c(j)$$

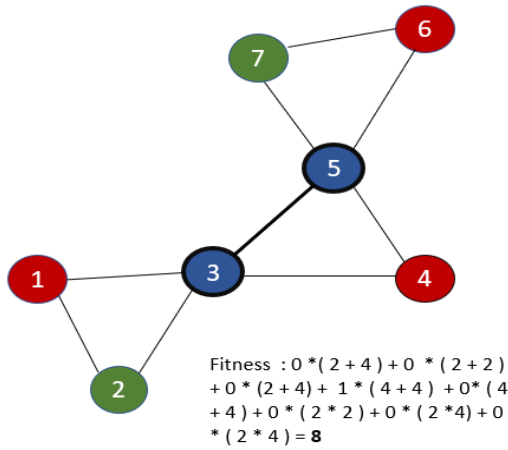


Fig 2.A Graph 3 coloring with one conflict, require at least 2 step to solve, fitness of 8

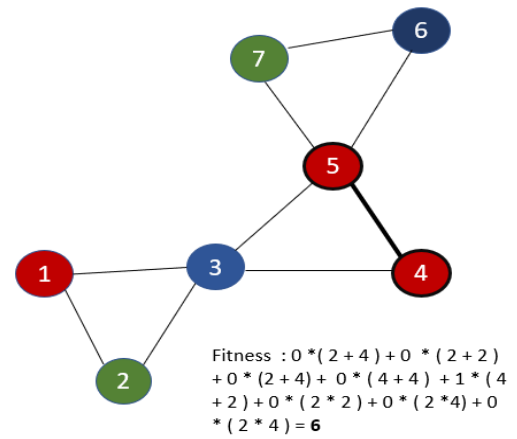


Fig 2.B Graph 3 coloring with one conflict, require 1 step to solve, fitness of 6

In this new fitness , once again , each edge will have a score based on whether it linked two vertices of the same color or not, multiplying by the sum of the degree of these vertices . The sum of each edge score, will allow us to discriminate between both configurations and preferentially selecting configuration B since it has the lowest fitness. The aim will be also to have a fitness of 0 , synonym of conflict free graph .

2. Random Generation of an instance of the problem

1. Input:

- Undirected graph (n nodes and e edges)
- $K = \max_degree$

The input will be an undirected graph with n nodes and e edges and a value k which stands for the maximum number of colors we will use to color the graph. We set k equals to the maximum degree of the graph. Because if we use the maximum degree as K, we can be sure that we can find a perfect solution (fitness score = 0).

2. Output: set of solutions

The output will be a set of solutions, because there may exist different coloring methods for one graph.

3. Variables: $X_v = \{X_1, X_2, X_3, \dots, X_n\}$

The variables in our problem will be a set of vertices X_v

4. Values: a list of integers in $\text{range}(0, k+1)$

The values will be a list of integers that present different colors.

If $X_v = 0$, it means the vertex is not colored; if $X_v \in \text{range}(1, k+1)$, it means the vertex is colored.

5. Constraints: $X_v \neq X_{v'} \quad \forall [v, v'] \in E$

The constraints will be that the two vertices can not have the same color if they are neighbors.

6. Initial state

The initial state is an uncolored graph, shown in Fig3A, so $X_v = \{0, 0, 0, \dots, 0\}$

7. Goal state

The goal state is a colored graph and each two neighboring vertices have different colors, for example, one of the goal states of Fig3.A is $X_v = \{4, 4, 1, 3, 3, 2, 1, 1, 3, 1\}$, shown in Fig3.B.

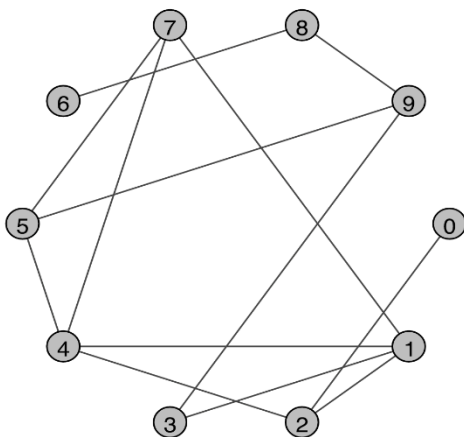


Fig 3.A: an example of initial state which is an undirected graph with 10 nodes, 17 edges

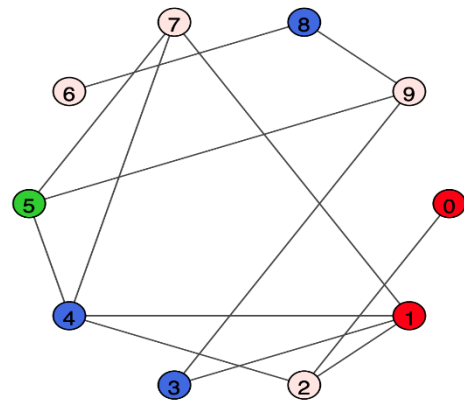


Fig3.B: an example of goal state uncolored which is a colored undirected graph with 10 nodes, 17 edges

3. Explanation of problem coding and the fitness function

3.1 Genetic Algorithm

3.1.1 Definition of Genetic Algorithm

A genetic algorithm (GA)¹ is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). In a GA, the evolution usually starts from a population of randomly generated individuals, and is an iterative process through mutation, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness score is usually the value of the objective function in the optimization problem being solved.

3.1.2 Six steps for solving this problem by using Genetic Algorithm

Step 1: Define a function to randomly generate an undirected graph with N nodes and e edges

Igraph² is used to generate the graph with N nodes and e edges are generated randomly. The input of the function is n and e , such as $N=10$, $e = 30$, and the output is a graph, shown in Fig4. In this step, it contains three small steps. Firstly, we use igraph to generate an undirected graph with nodes and 0 edges. Then we use `random.sample()` function to generate e pairs of nodes randomly without repetitions. Lastly, the e pairs of nodes are added to the graph as the edges. Taking the graph in Fig4 as an example, the `max_degree` of this graph is 8 and we will use maximum 8 colors to fill the nodes, so $K=8$.

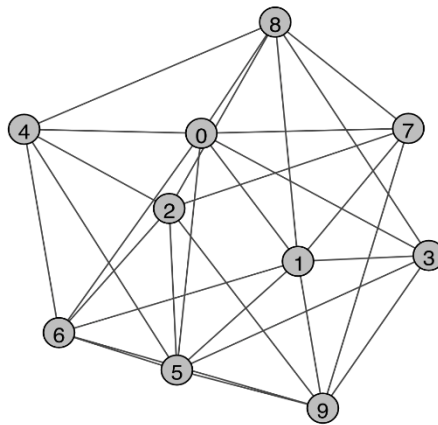


Fig4 An uncolored and undirected graph with 10 nodes and 30 edges

¹ Mirjalili, S. (2018) "Genetic algorithm In: Evolutionary Algorithms and Neural Networks," *Studies in Computational Intelligence*, pp. 43–55. Available at: https://doi.org/10.1007/978-3-319-93025-1_4.

² Csardi, G. and Nepusz, T., 2006. The igraph software package for complex network research. *InterJournal, complex systems*, 1695(5), pp.1-9.

Step 2: Define a function to create initial population

The input of the function is K and N, and the output is a N*N matrix (initial_population), which presents N initial solutions, is randomly generated. The value of each entity is in range 1 to K+1 and each value stands for a color. So, for $\forall(i, j) \in \{0, 1, 2, \dots, N\}$, $\text{initial_population}[i, j] \in \{1, 2, 3, \dots, K\}$.

In the matrix, each row is one potential solution, and we randomly generate N solutions as the initial population. So each solution will be a list of integers that are in $\{1, 2, 3, 4, \dots, K\}$

```
array([[3, 4, 2, 2, 2, 6, 2, 2, 3, 2],
       [5, 3, 6, 2, 6, 5, 4, 5, 7, 4],
       [3, 6, 6, 7, 1, 7, 5, 7, 7, 6],
       [2, 1, 7, 2, 7, 4, 7, 2, 7, 6],
       [4, 4, 5, 4, 7, 4, 4, 5, 1, 6],
       [3, 1, 5, 6, 4, 6, 1, 1, 5, 7],
       [2, 3, 6, 6, 6, 2, 1, 6, 5, 6],
       [6, 1, 5, 2, 2, 3, 5, 6, 3, 4],
       [6, 4, 5, 2, 3, 7, 6, 7, 2, 5],
       [5, 1, 6, 1, 1, 6, 1, 2, 1, 4]])
```

Fig5 A randomly generated N*N matrix which presents the initial population

Step 3: Define a fitness function with conditions to determine the optimal solution

After the initial population is generated, the next step is to rate each potential solution by using a fitness function and choose the best ones by using a selection function.

Fitness function (1): the number of edges which has two same colored vertices

1- Set fitness score1 =0

2- iterate of each edge

2.1- compare the color of source node and target node

2.2- if they have the same color, score1 will be added of 1

3- return score1

Fitness function (2):

1- Set fitness score2 =0

2- iterate of each edge

2.1- compare the color of source node and target node

2.2- if they have the same color, score2 will be added of source_node_degree and target_node_degree

3-return score2

For both fitness functions:

- 1) The solution with the lowest fitness score is considered as the optimal solution and our expected fitness score is 0, which means the solution is perfect.
- 2) The input of the fitness function will be our graph and initial population, and the output will be a list of scores for each potential solution in the initial population.

After calculating the fitness scores for each solution in the initial population in Fig5 by using fitness function (1) and (2) respectively, two lists of scores, scores1 and scores2, are generated, shown in Fig6.

```
scores1: [12, 2, 7, 8, 4, 3, 8, 2, 2, 6]
scores2: [156, 25, 87, 101, 41, 37, 107, 25, 26, 74]
```

Fig6 : Fitness scores of initial population. Scores1 is calculated by fitness function (1), and scores2 is calculated by fitness function (2)

Selection function:

We decided to choose the top 2 solutions from the initial population. Two lists are used to store the index of the rank 1 and rank 2 solutions respectively.

For example, in scores1, the lowest fitness score is 2 and there is not only one solution whose score is 2, all the solutions with a fitness score as 2 will be put in the top1 list. All the solutions which rank second will be put into the top2 list, shown in Fig7.

So the input of the selection function will be our graph and the fitness scores of the initial population, and the output will be two lists that contain the index of the good scores, top1 and top2.

```

in scores1:
top1:[1, 7, 8]
top2:[5]
in scores2:
top1:[1, 7]
top2:[8]

```

Fig7 The list of index of ranking No.1 and No.2 solutions by using scores1 and socres2 separately.

Step 4: define a crossover and mutation function

crossover function:

After using element crossover, single-point crossover and two-point crossover, we found that element crossover can find the solutions faster, so we choose element crossover for the crossover function.

- 1- Check if we already have the solution in top1 list
- 2- if the scores in the top1 list is 0, we find the solution
- 3- if the scores in the top1 list is not 0
 - 3.1- $top = top1 + top2$
 - 3.2- find all the combinations of two elements in the top list
 - 3.3- Each combination that presents two solutions (s1 and s2) will be used as the input of the crossover function. A point p will be randomly chosen and the element in point p in s1 will be exchanged with the point p in s2.
 - 3.3.1- The crossover function will not stop until the fitness score = 0 or the $nb_iteration = e * 10$
 - 3.3.2- all the solutions, the nb_iterations for finding a solution will be saved
 - 3.4 $min_iteration = \min(nb_iterations)$ and $avg_iteration = \frac{\sum(nb_iterations)}{\text{len}(nb_iterations)}$
 - 3.5 return all_solutions, min_iteration, avg_iteration, nb_solution

The improvement of this crossover function is that we use different combinations of the ranking 1 and ranking 2 initial populations to generate different offsprings. Because some combinations of the populations can find the solution very fast, such as the

initial_population1 and initial_population7, shown in Fig 8.A. The solution is found after 3 iterations of element crossover. Some combinations, such as initial_population1 and initial_population8, can not find the solution very fast, shown in Fig8.B.

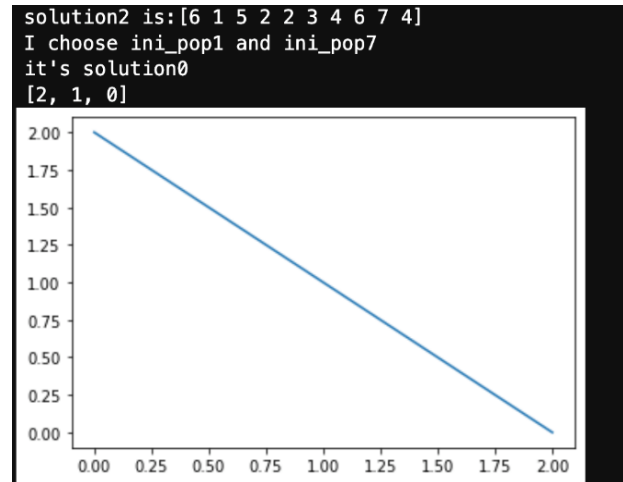


Fig8.A Use initial_population1 and 7 as input to do crossover with fitness function(1) and our algorithm finds the solution after 3 iterations. The first list is the solution, and the second list is the fitness score in each iteration. The x axis stands for the index of iterations and the y axis presents the fitness score.

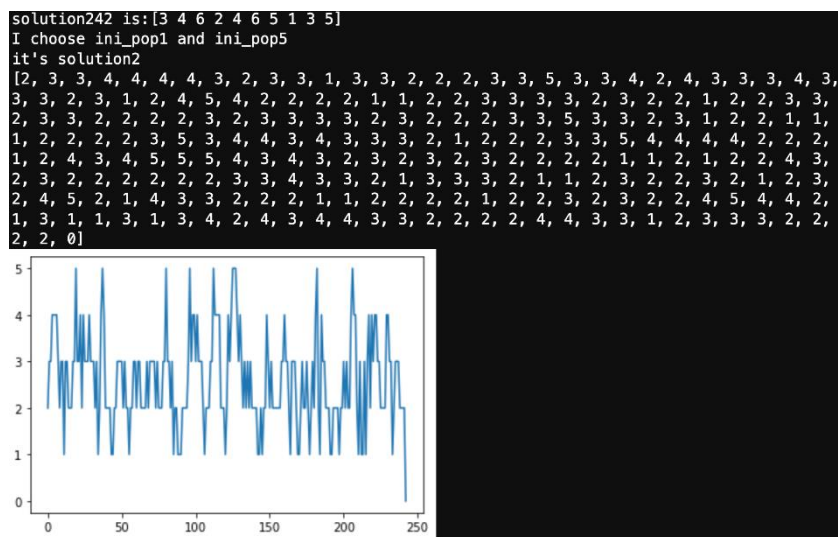


Fig8.B Use initial_population1 and 5 as input to do crossover with fitness function(1) and our algorithm finds the solution after 243 iterations. The first list is the solution, and the second list is the fitness score in each iteration. The x axis stands for the index of iterations and the y axis presents the fitness score.

Step 5: Set the genetic algorithm options, output and stopping criteria

This exchanging process will not stop until we find the best solution (fitness score =0) or the iteration time is more than $e * 10$. (e is number of edges)

Step 6: Define a plot function to plot the optimal solution on the graph

Once our algorithm finds the best solution, it will print the solution and plot the colored graph by using igraph, such as the graph shown in Fig9.

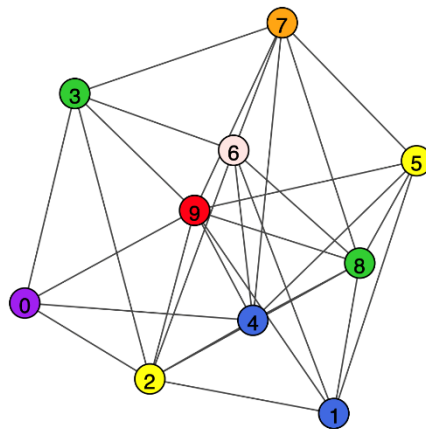


Fig 9 One of the solutions for the graph in Fig 4

3.2 Tabu Search

3.2.1 Definition of Tabu Search

The Tabu Search is a method who can be viewed as an iterative neighborhood approach, where the neighborhood changes dynamically. Tabu search (TS) explores a set of problem solutions, denoted by X, by repeatedly making moves from one solution s to another solution s' located in the neighborhood N(s) of s until a stopping criterion has been satisfied (fitness score = 0 or number of iteration exceeded) . TS enhances local search by actively avoiding points in the search space already visited. While Neighbouring solutions are found from the initial randomized solution. The best solution is selected, and is added to a tabu list. For future iterations, tabu items are disqualified as potential candidates, unless enough time has

passed and they can be reconsidered. This prevents the tabu search from getting stuck at a local minimum. And an aspiration criteria can be used to prematurely “free up” a tabu item and renew it for consideration.

3.2.2 Setting TABU algorithm

Step 1 : Define a function to randomly generate a $N \times N$ binary matrix and plot the undirected graph

We decided to use a specific library to generate an undirected graph , in this very case we used the library “networkx” which allows us to tune the graph with two parameters : 1) number of nodes ; 2) Probability for edge creation. Then a adjacent matrix have been generated from this graph

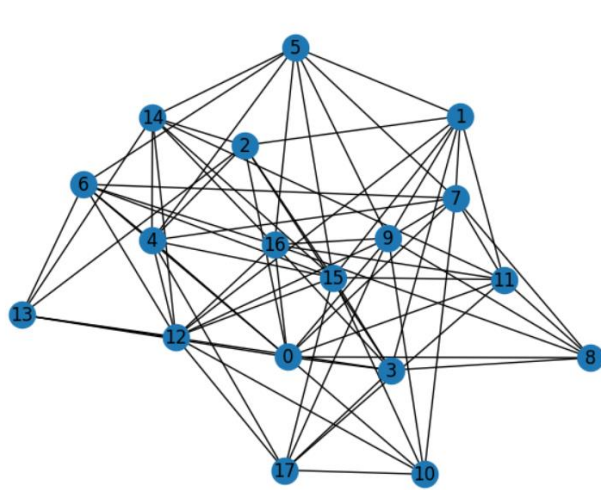


Fig 10.A : Nodes =18 , edges creation probability = 0.5

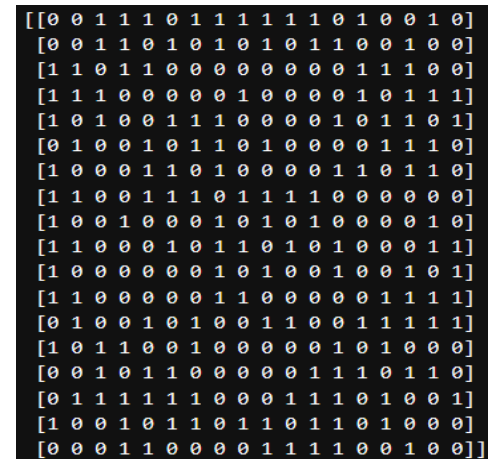


Fig 10.B Adjency matrix 18 x18

Step 2: Define a function to create initial solution

To define an initial solution we firstly choose the number of colors K wanted for coloring the graph. As an exemple, if $K = \text{max degree}$, then each node will be assigned randomly a value between 0 and max degree representing the color. Then an initial solution will be represented as the following, for each nodes 0 to i) in the graph \Rightarrow population : { nodes(1) :

colors(random(0,max degree)) , nodes(2) : colors(random(0,max degree)) ... nodes(i) : colors(random(0,max degree)) } .

Step 3: Define a fitness function with conditions to determine the optimal solution

The two fitness functions described have been used , and compared . The solution with the lowest fitness function score is considered as optimal and a correct K color , should have a fitness of 0 .

Fitness function:

The input of the fitness function will be our graph and initial solution, and the output will be a score for the given input solution. The fitness (1) is computed as follow :

- 1 - The number of conflicts ($\{V_i, V_j\} \in E, \text{color}(V_i) = \text{color}(V_j)$) is firstly set to 0
- 2 - for each two vertices (V_i, V_j) in the graph :
 - 3 - if $\{V_i, V_j\} \in E$ and $c(i) \neq c(j)$
 - 4 - Conflict s will be added of 1

As an exemple here is a typical output of the fitness function for a given graph through the language python, with a value computed for each vertices $V(0)$, $V(1)$... $V(17)$. The fitness value is 13 .

```
The fitness value for the solution {0: 1, 1: 2, 2: 0, 3: 2, 4: 3, 5: 2, 6: 3, 7: 1, 8: 0, 9: 0, 10: 2, 11: 1, 12: 3, 13: 0, 14: 0, 15: 1, 16: 0, 17: 0} is 13
```

Fig 11 Fitness (1) value of a given graph , with x :y , x correspond to a vertices and y correspond to the score related . Fitness is the sum of all scores

For the fitness (2) , the difference will reside on in the last step the function :

- 1 - Conflict s is set to 0
- 1.1 - degrees will be a set representing all degree for each vertices
- 2 - for each two vertices in the graph :
 - 3 - if $\{V_i, V_j\} \in E$ and $c(i) \neq c(j)$

$$4 - \text{Conflict } s = s + 1 * (\text{Degree}[\text{vertices}(i)] + \text{Degree}[\text{vertices}(j)])$$

The fitness value for the solution {0: 1, 1: 2, 2: 0, 3: 2, 4: 3, 5: 2, 6: 3, 7: 1, 8: 0, 9: 0, 10: 2, 11: 1, 12: 3, 13: 0, 14: 0, 15: 1, 16: 0, 17: 0} is 219

Fig 12 Fitness (2) value of a given graph , with x :y , x correspond to a vertices and y correspond to the score related . Fitness is the sum of all scores

This new fitness will put a bigger “sanction” for same color vertices with a higher degree

Step 4: find all the neighbors of initial solution: Swap

Once we have the initial solution , we need to produce a different candidate solution from our current solution S. The function to produce Neighbors of our initial solution will be computed as follow:

For exemple, for $K = 2$, the colors will be a set of integer from 0 to 2, inside our solution $S = \{ \text{vertices}(0): \text{random color}(k) \dots \text{vertices}(i) : \text{random color}(k) \}$.

- 1 - Choose one random vertices (V_i) of our initial solution $S = \{ 0 : 1 , 1 : 0 , 2 : 1 \}$
- 2 - Choose one color randomly $c \Rightarrow$ lets take 1
- 3 - if color of random vertices (V_i) = random color (y) :
 - 3.1 - swap last color of vertices with current color (y)
- 5 - add new color (y) of random vertices (i)

Then the new solution will be : $\{ 0 : 0 , 1 : 0 , 3 : 1 \}$, the color of 0 has been changed from 1 to 0 .

here is a neighbor_solutions : {0: 1, 1: 2, 2: 0, 3: 2, 4: 3, 5: 2, 6: 3, 7: 1, 8: 0, 9: 0, 10: 2, 11: 1, 12: 3, 13: 0, 14: 0, 15: 0, 16: 0, 17: 0} of the previous one : {0: 1, 1: 2, 2: 0, 3: 2, 4: 3, 5: 2, 6: 3, 7: 1, 8: 0, 9: 0, 10: 2, 11: 1, 12: 3, 13: 0, 14: 0, 15: 1, 16: 0, 17: 0}

Fig 13 : new neighbor solution using swapping

Step 5 - Compare the neighbors' fitness score with the initial solution

We will compare our fitness score between the initial and neighbor's solution . In order to make the comparison , we will create a variable `new_conflict` , which will represent the fitness score for the new solution, and compare it with the variable `conflict_count` , who represents the fitness of the initial solution.

- 1 - `New_conflict = 0`
Do fitness function ...
- 2 - if `New_conflict < conflict_count` :
 - we found an improved solution

Step 6 - Choose the best one from neighbor and put the initial one to the tabu list

The tabu tenure is set to 7, but different sizes have been tested since the its size has a great impact on the Tabu search performance. Also , as it might be common to store inside the tabu list each candidate solution. Since our problem can require a high number of nodes to be taken into account, it sounds very consuming to store the entire solution inside the tabu list . Furthermore when our size is equal to 7. Instead , we will store the Nodes that have been swapped, so in this case the combination of only one key-value pair { Node , Color } will be stored at a time rather than the entire solution (which is a set of {Node : Color}) . As an hypothetical exemple, after 2 iterations , our tabu list will be : `Tabu = [{0,2} , {1,0}]` , which represents a combination of 2 { nodes , color} . It means that the combination for nodes 0 with color 2 and nodes 1 color 0 cannot be swapped for the next 7 iterations . The tabu initialization is described as follow :

- 1- initialize tabu as empty queue
.....
- 2 - select one node that have been swapped from a candidate solution
- 3 - choose node : color other than current (step 4) and add to tabu queue
...
- 4 - if length of tabu queue > length tabu size (7) :
 - 4.1 - remove tabu list oldest element

The list used to record the recent solutions will prevent them from reoccurring for a specified number of iterations , which helps the search to move away from previously visited solutions thus performing more extensive exploration.

Step 7 : Define Aspiration criteria

If the solution found in the current iteration has a value (fitness function value) better than the currently-known best solution's value, but the move is Tabu, we can use the Aspiration Criteria to override the tabu state of this move, thereby including the otherwise-excluded solution in the allowed set.

In our case , the aspiration Criteria will be based on the fitness value (conflict_count and new_conflict , representing respectively the fitness score for the initial solution and the neighbor solution).

- 1 - initialization of the aspiration variable, set to an empty queue .
- 2 - If $\text{new_conflict} < \text{conflict_count}$ (found an improved solution) :
 - 3 - $(\text{conflict_count} - 1)$ is added to Aspiration queue
 - 4 - if $\text{new_conflict} \leq \text{aspiration}$:
 - 5 - aspiration will replace $(\text{conflict_count} - 1)$ by (
 $\text{new_conflict} - 1$)
 - 6 - If key-value pair nodes : Colors of neighbors in tabu list :
 - 7 - remove nodes and colors from tabu

Indeed, the aspiration criteria will allow us to bypass the tabu list , if the score of the neighbor solution is far better than the initial solution . It allows us to find the optimal solution faster .

```
33 -> 32
32 -> 16
32 -> 17
tabu permitted; 17 -> 16
36 -> 34
tabu permitted; 34 -> 17
17 -> 0
Found coloring:
{0: 3, 1: 3, 2: 1, 3: 4, 4: 0, 5: 4, 6: 1, 7: 2, 8: 1, 9: 0, 10: 1, 11: 0, 12: 4, 13: 2, 14: 3, 15: 2, 16: 2, 17: 3}
```

Fig 14 : exemple of aspiration criteria inside tabu algorithm (tabu permitted)

TABU algorithm output

The tabu algorithm developed was successful in finding the solution for Fig10.A. as depicted below.

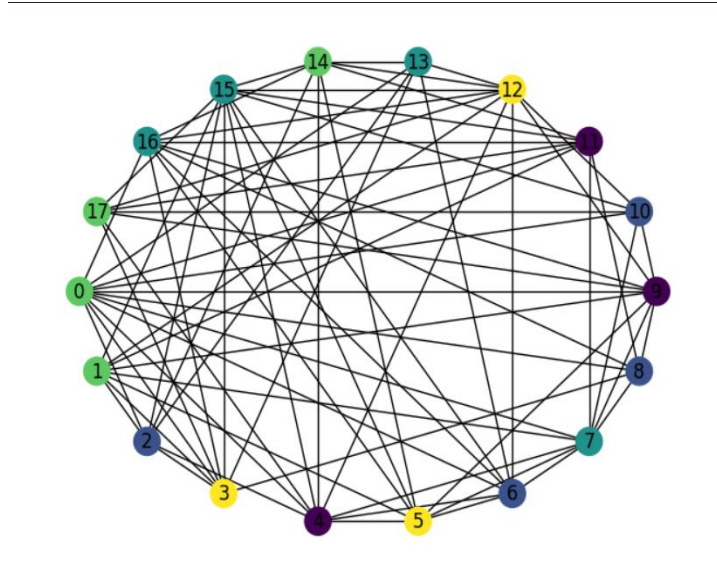


Fig 15 : K colorable graph with $K=5$, for graph with 18 vertices .

4. Experimental evaluation

4.1 Comparison in two Fitness functions

4.1.1 Two Fitness functions comparison by using Genetic Algorithm

If we take the graph in Fig4 as an example, the algorithm finds the solution after 58 iterations when we use initial_population1 and 8 to do crossover with fitness function(1), but it can find the solution in 12 iterations if fitness function(2) is used, shown in **Fig16** A,B .

In **Fig17** A,B in 243 iterations, the solution is finally found with the fitness function(1), while 2 iterations are needed with fitness function(2).

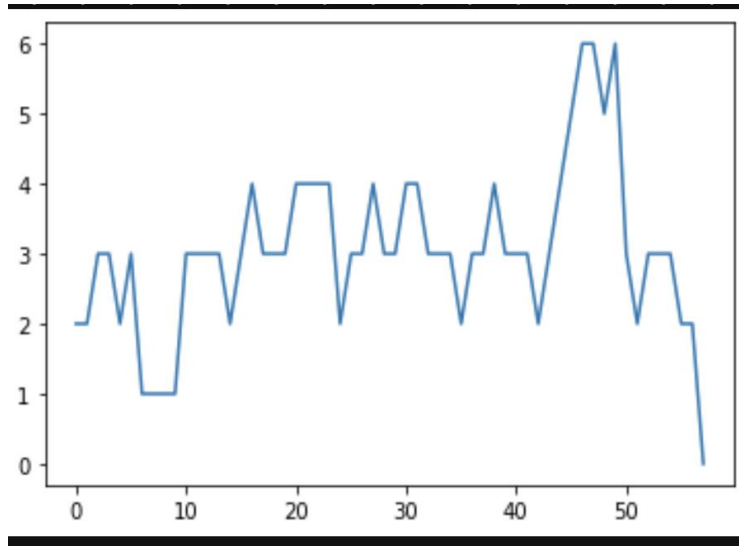


Fig16.A The graphs show the number of iterations for finding the solution for the graph in Fig4 if we use initial_population1 and 8, **using fitness (1)**
The x axis stands for the index of iterations and the y axis presents the fitness score

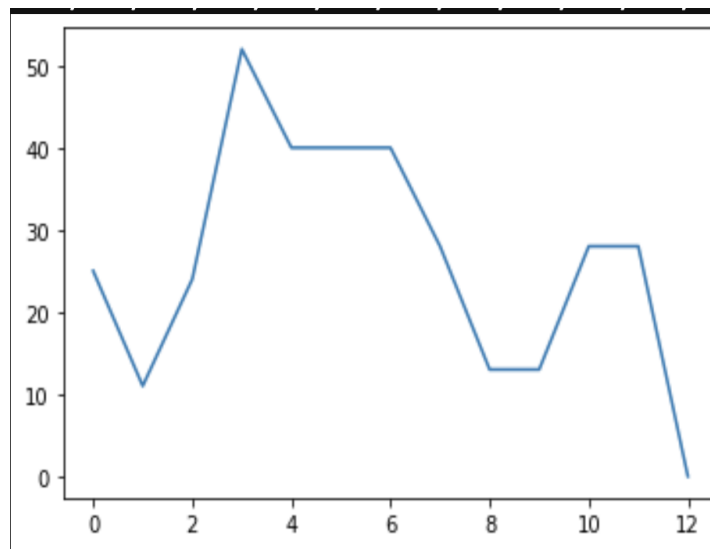


Fig16.B The graphs show the number of iterations for finding the solution for the graph in Fig4 if we use initial_population1 and 8, **using fitness (2)**
The x axis stands for the index of iterations and the y axis presents the fitness score

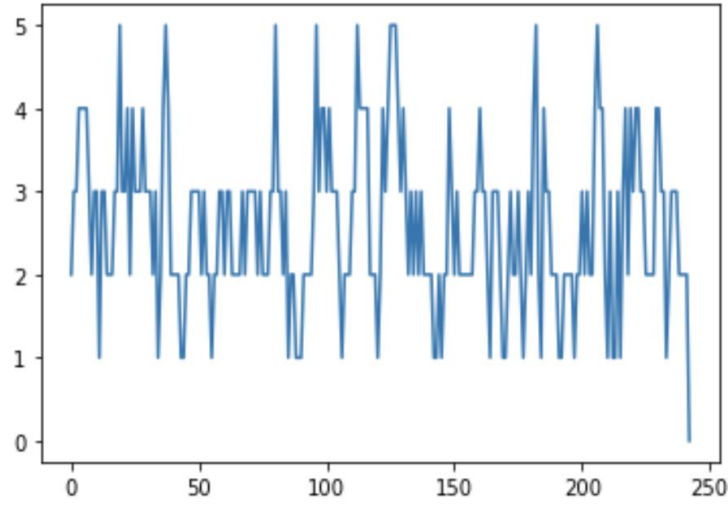


Fig17.A The graphs show the number of iterations for finding the solution for the graph in Fig4 if we use initial_population1 and 5 **using fitness (1)** . The x axis stands for the index of iterations and the y axis presents the fitness score.

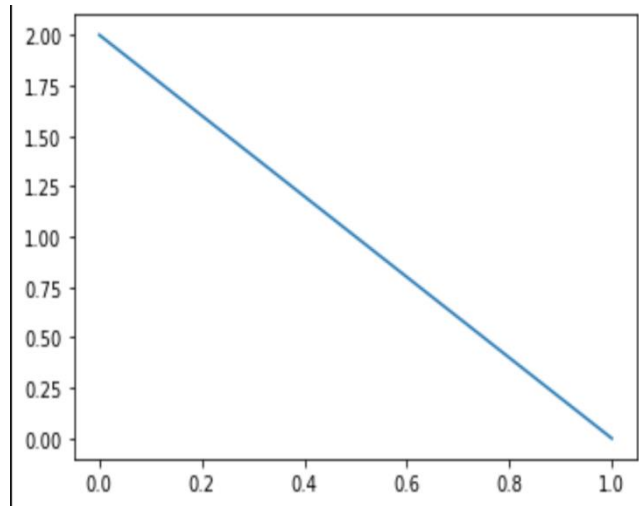


Fig17.A The graphs show the number of iterations for finding the solution for the graph in Fig4 if we use initial_population1 and 5 **using fitness (2)** . The x axis stands for the index of iterations and the y axis presents the fitness score.

100 different graphs with 10 nodes and 30 edges are used in our algorithm, and the average minimum iterations, average of average iterations for each graph for finding the solution and the average number of solutions are compared. After the 100 runs, we find that the fitness function(2) has better performance than fitness function (1) based on the speed of finding a solution, shown in Fig18 .

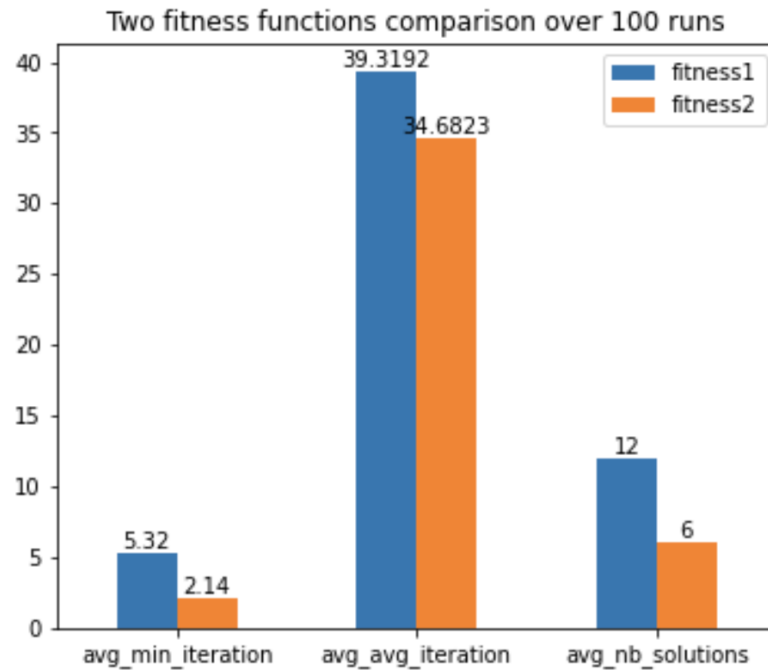


Fig18 The comparison of two fitness functions over 100 runs by using 100 graphs with 10 nodes and 30 edges.

4.1.2 Two Fitness functions comparison by using Tabu Search

As said both fitness have been used in order to compare their respective efficacy . For this graph , the first fitness found the correct solution after 41 iteration , where the second fitness found the correct solution after 24 iterations .

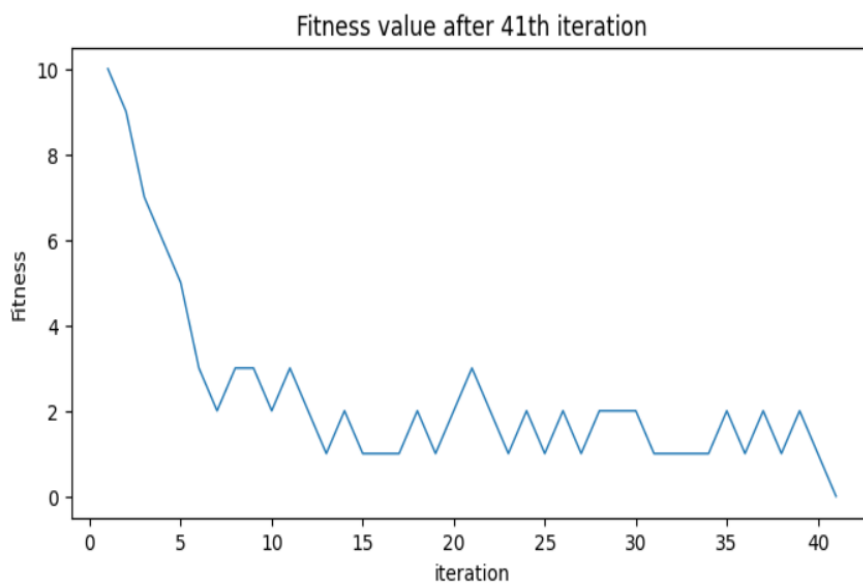


Fig 16.A Variation of fitness value during tabu algorithm for fitness (1)

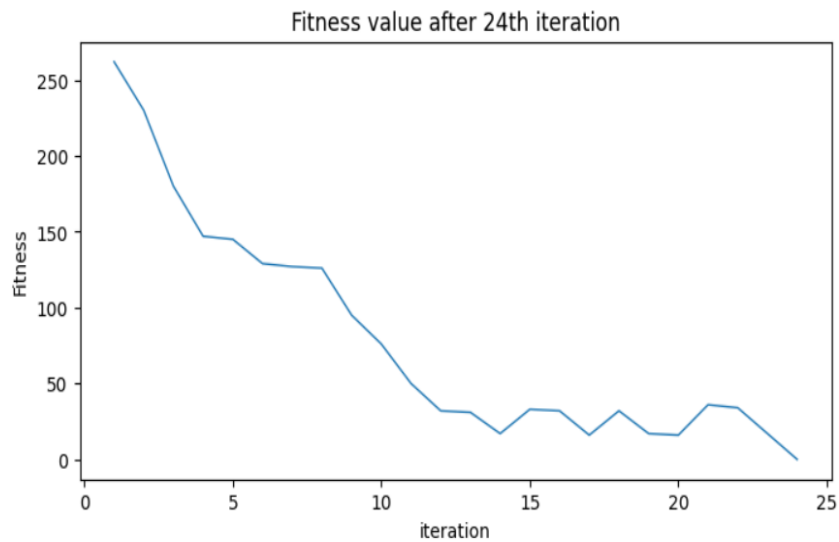


Fig 16.B Variation of fitness during tabu algorithm for fitness (2)

While such difference seems at first significative , such variation can happen for the same fitness within the same graph . In order to be sure that the new fitness function actually improve the algorithm by reducing the amount of iteration needed to found a fitness = 0 , the tabu algorithm have been run 1000 times for the same graph in fig10.A and the mean number of iteration have been computed for both fitness.

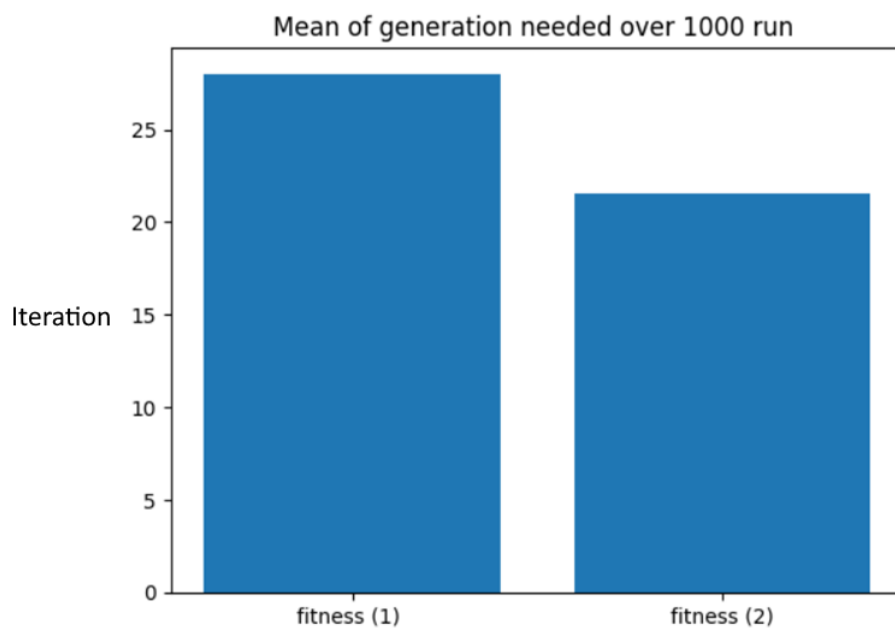


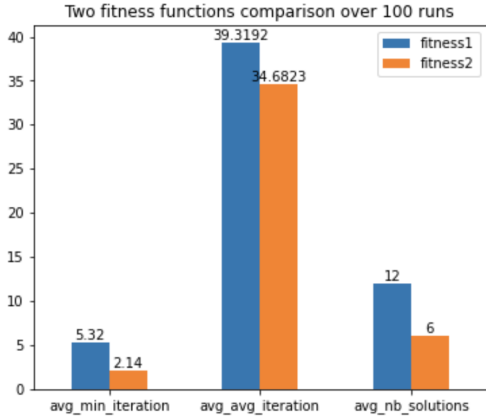
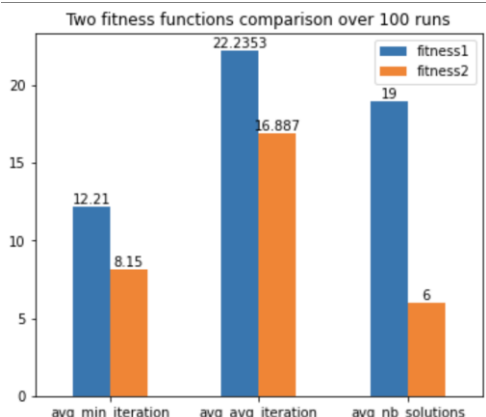
Fig 17 comparison of both fitness for graph 10.A for 1000 run of the TABU algorithm

The fitness 2 who taking into account the degree of vertices need less generation to found the solution , comparing to the fitness 1. Such difference highlight the importance of the fitness function in order to improve such algorithm .

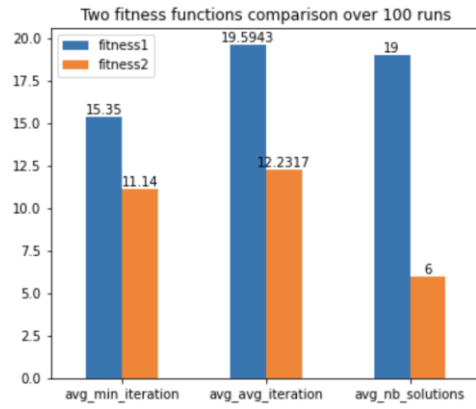
4.2 Comparison in small, medium and large graphs

4.2.1 Comparison in different graphs by using Genetic Algorithm

Table 1 The results of K-color graph problem in different size of graph and the execution time

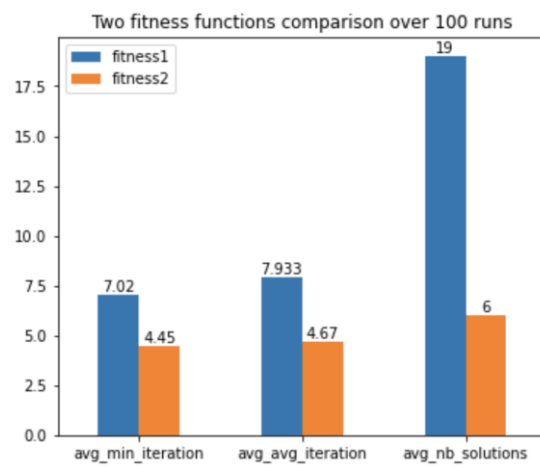
Input Graph	Fitness comparison during 100 runs	Average Execution time for 1 run during 100 runs												
A graph with 10 nodes and 30 edges	 <table border="1"> <caption>Two fitness functions comparison over 100 runs (10 nodes, 30 edges)</caption> <thead> <tr> <th>Metric</th> <th>fitness1</th> <th>fitness2</th> </tr> </thead> <tbody> <tr> <td>avg_min_iteration</td> <td>5.32</td> <td>2.14</td> </tr> <tr> <td>avg_avg_iteration</td> <td>39.3192</td> <td>34.6823</td> </tr> <tr> <td>avg_nb_solutions</td> <td>12</td> <td>6</td> </tr> </tbody> </table>	Metric	fitness1	fitness2	avg_min_iteration	5.32	2.14	avg_avg_iteration	39.3192	34.6823	avg_nb_solutions	12	6	0.025 s
Metric	fitness1	fitness2												
avg_min_iteration	5.32	2.14												
avg_avg_iteration	39.3192	34.6823												
avg_nb_solutions	12	6												
A graph with 100 nodes and 500 edges	 <table border="1"> <caption>Two fitness functions comparison over 100 runs (100 nodes, 500 edges)</caption> <thead> <tr> <th>Metric</th> <th>fitness1</th> <th>fitness2</th> </tr> </thead> <tbody> <tr> <td>avg_min_iteration</td> <td>12.21</td> <td>8.15</td> </tr> <tr> <td>avg_avg_iteration</td> <td>22.2353</td> <td>16.887</td> </tr> <tr> <td>avg_nb_solutions</td> <td>19</td> <td>6</td> </tr> </tbody> </table>	Metric	fitness1	fitness2	avg_min_iteration	12.21	8.15	avg_avg_iteration	22.2353	16.887	avg_nb_solutions	19	6	0.056 s
Metric	fitness1	fitness2												
avg_min_iteration	12.21	8.15												
avg_avg_iteration	22.2353	16.887												
avg_nb_solutions	19	6												

A graph with 1000 nodes and 5000 edges




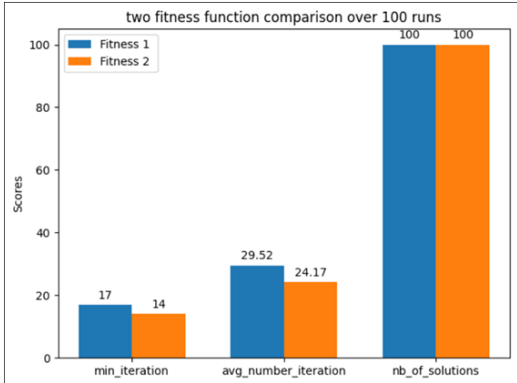
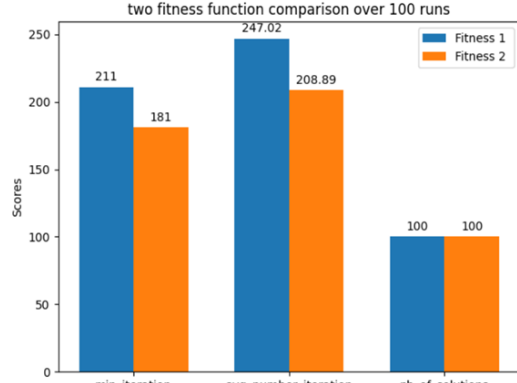
5.38 s

A graph with 1000 nodes and 9000 edges



10.20 s

4.2.1 Comparison in different graphs by using Tabu Search

Input Graph	Fitness comparison during 100 runs	Average Execution time for 1 run during 100 runs												
A graph with 10 nodes and 30 edges	 <table border="1"> <caption>Data for Fitness Comparison (10 nodes, 30 edges)</caption> <thead> <tr> <th>Metric</th> <th>Fitness 1</th> <th>Fitness 2</th> </tr> </thead> <tbody> <tr> <td>min_iteration</td> <td>2</td> <td>2</td> </tr> <tr> <td>avg_number_iteration</td> <td>4.16</td> <td>3.29</td> </tr> <tr> <td>nb_of_solutions</td> <td>100</td> <td>100</td> </tr> </tbody> </table>	Metric	Fitness 1	Fitness 2	min_iteration	2	2	avg_number_iteration	4.16	3.29	nb_of_solutions	100	100	Fitness 1 : ~0.14s Fitness 2 : ~0.14s
Metric	Fitness 1	Fitness 2												
min_iteration	2	2												
avg_number_iteration	4.16	3.29												
nb_of_solutions	100	100												
A graph with 100 nodes and 500 edges	 <table border="1"> <caption>Data for Fitness Comparison (100 nodes, 500 edges)</caption> <thead> <tr> <th>Metric</th> <th>Fitness 1</th> <th>Fitness 2</th> </tr> </thead> <tbody> <tr> <td>min_iteration</td> <td>17</td> <td>14</td> </tr> <tr> <td>avg_number_iteration</td> <td>29.52</td> <td>24.17</td> </tr> <tr> <td>nb_of_solutions</td> <td>100</td> <td>100</td> </tr> </tbody> </table>	Metric	Fitness 1	Fitness 2	min_iteration	17	14	avg_number_iteration	29.52	24.17	nb_of_solutions	100	100	Fitness 1 : ~ 0.67 Fitness 2 : ~0.58
Metric	Fitness 1	Fitness 2												
min_iteration	17	14												
avg_number_iteration	29.52	24.17												
nb_of_solutions	100	100												
A graph with 1000 nodes and 5000 edges	 <table border="1"> <caption>Data for Fitness Comparison (1000 nodes, 5000 edges)</caption> <thead> <tr> <th>Metric</th> <th>Fitness 1</th> <th>Fitness 2</th> </tr> </thead> <tbody> <tr> <td>min_iteration</td> <td>211</td> <td>181</td> </tr> <tr> <td>avg_number_iteration</td> <td>247.02</td> <td>208.89</td> </tr> <tr> <td>nb_of_solutions</td> <td>100</td> <td>100</td> </tr> </tbody> </table>	Metric	Fitness 1	Fitness 2	min_iteration	211	181	avg_number_iteration	247.02	208.89	nb_of_solutions	100	100	Fitness 1 : ~2102s Fitness 2 : ~1890s
Metric	Fitness 1	Fitness 2												
min_iteration	211	181												
avg_number_iteration	247.02	208.89												
nb_of_solutions	100	100												
A graph with 1000 nodes and 9000 edges	<p>Too much time , Computer not efficient enough</p>													

4.3 Comparison in Genetic Algorithm and Tabu Search

After comparing the Genetic Algorithm and Tabu Search, we found that Tabu Search can find more solutions than Genetic Algorithm in the same running iterations. For small graphs, Tabu search can find the solution faster, because the minimum number of iterations and average number of iterations that we can find the solution are smaller. However, for the bigger graph, Tabu Search will need more iterations to find the solution, and the performance of the Genetic algorithm is quite good and stable.

5. Conclusion

Genetic Algorithm and Tabu Search are used to solve the k-color graph problem. In order to improve the algorithms, we improved our fitness function successfully. We also improved a selection function for Genetic Algorithm by choosing multiple top1 and top2 solutions from initial populations, and using all the combinations of the top1 and top2 solutions to make crossover. Finally, we used different graphs which contain small size, medium size and big size. We find that Tabu Search has better performance in small size graphs, but with the increasing size, the performance decreased, while Genetic Algorithms have more stable and better performance for the big size graph.

6. Program in appendix

the code is original it has been realised by ourselves , it is available on github :
<https://github.com/GAOJinmei/Genetic-Algorithm-and-Tabu-Search-for-K-Graph-Coloring-Problem.git>

7 . References

- .Mirjalili, S. (2018) “Genetic algorithm In: Evolutionary Algorithms and Neural Networks,” *Studies in Computational Intelligence*, pp. 43–55. Available at: https://doi.org/10.1007/978-3-319-93025-1_4.
- .Csardi, G. and Nepusz, T., 2006. The igraph software package for complex network research. *InterJournal, complex systems*, 1695(5), pp.1-9.

.Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, [“Exploring network structure, dynamics, and function using NetworkX”](#), in [Proceedings of the 7th Python in Science Conference \(SciPy2008\)](#), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

.Hertz, A., de Werra, D.: Using tabu search techniques for graph coloring. *Computing* 39(4), 345–351 (1987)

.Rodriguez-Tello, E., Hao, J.K.: On the role of evaluation functions for heuristic search (working paper, 2007)

.Leighton, F.T.: A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards* 84(6), 489–503 (1979)

. Dorne, R., Hao, J.K.: Tabu search for graph coloring, T-colorings and set Tcolorings. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, 77–92 (1998)

.Eiben, A.E., van der Hauw, J.K.: Adaptive penalties for evolutionary graph coloring. In: Rao, A., Singh, M.P., Wooldridge, M.J. (eds.) *ATAL 1997*. LNCS, vol. 1365, pp. 95–106. Springer, Heidelberg (1998)

. Katoch, S., Chauhan, S.S. & Kumar, V. A review on genetic algorithm: past, present, and future. *Multimed Tools Appl* **80**, 8091–8126 (2021). <https://doi.org/10.1007/s11042-020-10139-6>