A genetic algorithm on the graph coloring problem

Pedro Martins 54390

Faculdade de Ciências e Tecnologia, Universidade do Algarve

ABSTRACT

In this article I present a genetic algorithm that solves the graph coloring problem. Some operators of crossover, mutation and selection were implemented to determine which ones work better. The algorithm is tested some graph instances in DIMACS format. The results were obtained, and an analysis of the results is given. The proposed genetic algorithm succeeded in giving approximate solutions to these instances, while some that are more complicated, it failed.

1 INTRODUCTION

The graph coloring problem is the following: Given any graph with *v* vertices and *e* edges, determine the colors that should be given to each edge, such that no neighbor of *e* has the same color of *e.* In other words: Given any graph *G*, does *G* admit a proper vertex coloring with *k* colors? This problem may appear simple at the first glance. But it's not. In fact, it's time complexity is so big that it is a NP-Complete problem, because we can check if the solution that we got "works", in polynomial time. This has the following running time:

$$\boldsymbol{O(2^e e)}$$

If we had a graph with $e = 500$, which is very very small to graphs that represent real world scenarios, we would get an astronomical number that no computer is able to solve with a regular-type algorithm. This is a classic example where we should try to use genetic algorithms (or other stuff).

This type of problems is impossible to solve in polynomial time. The number of different colors to paint the vertices of a graph is called the chromatic number $\chi(G)$. Effectively being able to solve this problem has a lot of applications in graph theory, and in real word scenarios like scheduling problems, radio frequency assignment and apparently sudoku as well.

It seems this problem is already complicated enough. But the problem is even harder, if we instead of trying to see if we can color with *k* colors we try to find the minimum number of colors. If we try this, we are no longer in a NP-Complete problem, but rather in a NP-Hard problem. This type of problems is even harder because we cannot check if a solution is the best solution possible in polynomial time. This "checking" procedure would also take us NP time.

2 My implementation

To solve this problem, I used a genetic algorithm that we previously had studied in class. I created a program in python (3.7). To create the graph structure, I used a library in python called *networkx,* that creates the underlying graph structure, and well it's useful instead of programming everything. To visualize the results I used *matplotlib.*

I implemented the following operators:
Crossover: uniform crossover, onepoint crossover, cycle crossover.

Selection: truncation selection, RWS, SUS, tournament selection with and without replacement.

For mutation it was a little different. It was a bit tricky to develop good mutation operators for this because it's not as simple as the onemax problem where we could just flip the bits. So I discovered three methods for this.

Mutation1:

```
foreach(edge in graph) {
        if(edge->color has the same color as an adjacent edge) {
                adjacentColors = all colors adjacent to edge;
                validColors = allColors except
        adjacentColors;
                newColor = random color from validColors;
                setColor(edge, newColor)
        }
}
```

Mutation2:

```
foreach(edge in graph) {
        if(edge->color has the same color as an adjacent edge) {
                newColor = random color from allColors;
                setColor(edge, newColor)
        }
}
```

Mutation3, simply change color of each vertex to a random color.

This were the operators developed. Now we need a good fitness function for this. And this is also tricky to do. Because it should be fast to compute and also good for the algorithm.

Our goal is to minimize the number of wrong edges connected. To do this, we can count the number of correctly connected edges subtracted by the number of wrong edges. But we can do variations of this, like giving more importance to the correct edges with a factor of *alfa*. Something like this.

An individual from this problem is represented as an *networkx* object with the following attributes:

- Adjacency list
- Fitness
- List of colors
- Graph *G*, containing the adjacency list and the colors for each node.

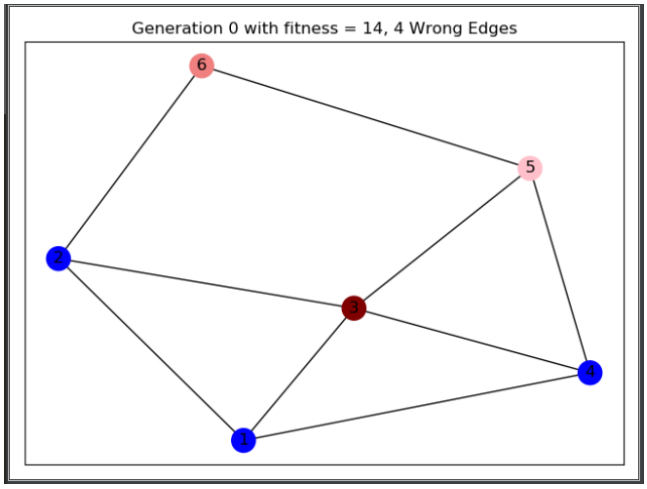
And my algorithm is simply changing the colors for each node, since the adjacency list is the same for each graph, otherwise it wouldn't make any sense.
Here is the sequence from the program.

1. Supply configuration info from the settings.txt file (such as p of crossover, population size and so on.
2. Read graph from txt file.
3. Create initial graph and with that graph create *k* individuals.
4. Start the normal sequence of a genetic algorithm with *i* iterations.
5. Visualize the final graph result with each color.

2.1 Example of one iteration

We start with a graph with 6 vertices and 9 edges. [(1,2), (2,3), (3,4), (4,1), (5,3), (6,2), (5,4), (3,1), (6,5)] And we want a solution with 4 colors.

First thing is we assign a random color to each edge. This is our generation 0.

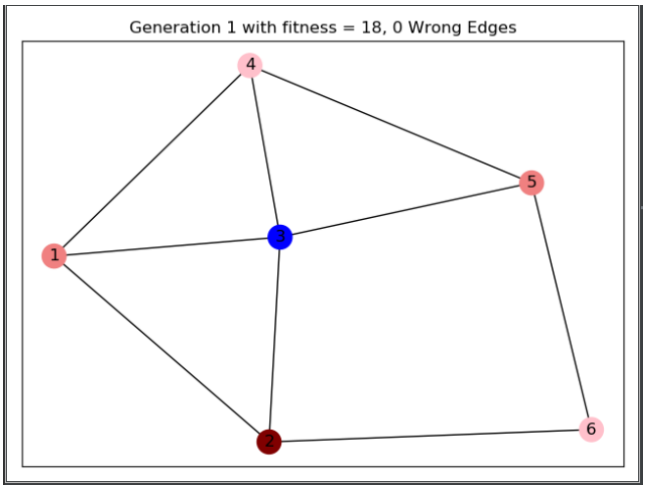


Next, we do tournament selection between 2 elements of the population and select the ones with higher fitness.

We then do uniform crossover, which is essentially just changing the color of each edge for the color of an edge of other graph with probability 0.5.

We then do mutation as presented in "mutation1".

Then we print the best individual from the population. If we have 0 wrong edges we can stop the algorithm.



In this case we got already optimum result, so we stop the algorithm

3 Obtained Results

The following results were obtained with the following graphs (vertices, edges):

1. sample_graph0 (95, 755)
2. sample_graph1 (11,20)
3. sample_graph2 (191, 2360)
4. sample_graph3 (23, 71)
5. sample_graph4 (47, 236)
6. sample_graph5 (25, 160)

All graphs except sample_graph5 are triangle-free graphs (undirected graph in which no three vertices form a triangle of edges). This type of graphs are the hardest to solve.

The tests were run on a xiaomi mi notebook pro with this specs:

i5-8250CPU 1.60GHZ
8GB RAM

The probabilities for crossover and mutation for all tests are:

P crossover: 0.99
P mutation: 0.7

Here are the more relevant tests. I did more tests, but the results are self-evident with these four tests.

Crossover: Uniform Crossover
Selection: Tournament Selection with replacement (k = 2)
Mutation: mutation1

Graph	Expected	Obtained	PopSize	Generations	Time
Graph 0	7	7	500	61	110.9 s
Graph 1	4	4	20	7	0.22s
Graph 2	8	12	500	94	479.8 s
Graph 3	5	5	70	28	1.07s
Graph 4	6	6	150	43	8.06s
Graph 5	5	6	150	39	4.64s

Crossover: Uniform Crossover
Selection: Tournament Selection without replacement (k = 2)
Mutation: mutation1

Graph	Expected	Obtained	PopSize	Generations	Time
Graph 0	7	7	500	66	121.64 s
Graph 1	4	4	20	3	0.21s
Graph 2	8	12	500	96	498.89 s
Graph 3	5	5	70	22	0.89s
Graph 4	6	6	150	42	7.14s
Graph 5	5	7	150	44	5.15s

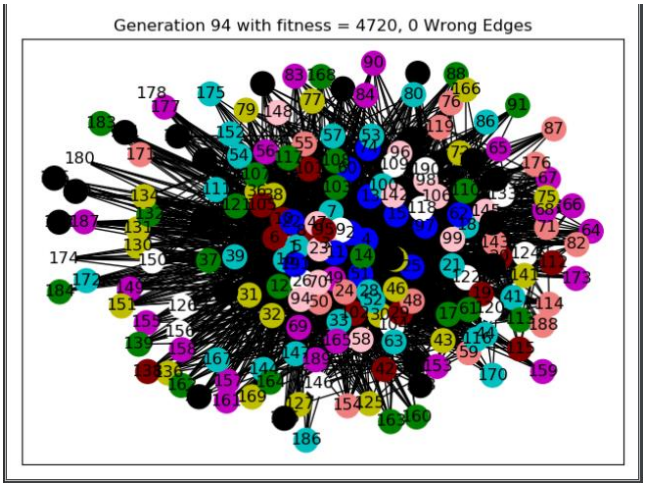
Crossover: Uniform Crossover
Selection: Truncation selection (p = 0.2)
Mutation: mutation1

Graph	Expected	Obtained	PopSize	Generations	Time
Graph0	7	11	500	89	90.1s
Graph1	4	4	20	3	0.18s
Graph2	8	---	---	---	---
Graph3	5	7	150	7	0.84s
Graph4	6	7	150	16	2.47s
Graph5	5	8	150	12	1.31s

Crossover: Cycle Crossover
Selection: SUS selection
Mutation: mutation2

Graph	Expected	Obtained	PopSize	Generations	Time
Graph0	7	---	---	---	---
Graph1	4	4	250	3	2.57s
Graph2	8	---	---	---	---
Graph3	5	6	250	19	3.89s
Graph4	6	---	---	---	---
Graph5	5	---	---	---	---

Here is graph0 solved. (what a mess…)



4 Discussion of the obtained results

The results clearly show that for this given implementation, the algorithm performed better with uniform crossover and mutation1. As for the selection method, it was more difficult to evaluate. There are differences between tournament selection with/without replacement and truncation selection. We can argue that these differences can be neglected since we didn't do a thorough statistical analysis and we didn't run it many times. But in my opinion, tournament selection is the winner. Now if it is with or

without selection is a more different question. I would say, we would need to do more tests.

Truncation selection gets stuck a lot in a local optimum. The reason being that if we always choose the same amount of elements, like the same chunk of elements all over again, the algorithm will try to converge as fast as possible. This can be seen when running the algorithm, truncation selection converges way faster than tournament selection but it will converge at the local optimum. A way to tackle this issue could be to change the percentage in truncation selection at runtime. In other words, we could adjust the *p* value of truncation selection at each iteration.

Also, I realized that this algorithm performs well when we have a very high crossover probability. Low probability showed that it gets stuck in a local optimum. But we also need a high mutation probability (a smart mutation) for us to reach the global optimum.

The algorithm also had very good results, except for the case of the sample_graph2, this is because this graph is much bigger than the others. In order to achieve better results, I could have switched algorithms during runtime. That is, within certain conditions, change the crossover, mutation or selection method to a different one.

During execution of the algorithm, since we have a very high mutation probability, the fitness is oscillating a lot in the beginning, but this is good, since this seemingly worsening fitness is what partly causes some solutions to be found.

Also, increasing the population size is the main factor to be able to reach a solution. The reason being, that if we start with a lot of individuals, we will have a lot of genetic diversity in our population. We can search for the global optimum in a lot of directions and be able to reach a solution more easily. This has the downside of making the algorithm much slower. It is arguably what makes it difficult to solve large graphs. For large graphs we need a different approach to this problem, probably developing better operators, and overall a strategy that changes the operators during runtime.

Regarding the tournament size, I did some experimentation and reached the conclusion k=2 is the best. If we do tournament with a lot of individuals we are giving room for the algorithm to find new solutions. Therefore we give the algorithm a lot of liberty to find optimum solutions in a lot of directions. If we did a tournament (in the limit) of the size of the population we would get the best individual from this generation. Yes, it has the highest fitness but this fitness may not be a good indicator of a good individual. We need to give chances to a lot of individuals, and then with mutation we speed the process of finding solutions.

One point crossover and cycle crossover shown to yield worse results than uniform crossover (surprisingly). Why this happens, no idea. Uniform crossover probably mixes each graph equally and gets the properties of both graphs in a fair way.

5 Conclusion

This problem was solved with a regular type genetic algorithm. It was shown that for large graphs the algorithm didn't perform well, since the complexity (even for a genetic algorithm) becomes large. This algorithm effectively solved most of the instances proposed, and with reasonable results. The main objective was achieved: solving a NP problem with a genetic algorithm.

ACKNOWLEDGMENTS

This work was done for the course of Evolutionary Computation for the master's in computer science, supported by the University of Algarve in 16 of December of 2019.

REFERENCES

[1] Graph Coloring Instances, https://mat.tepper.cmu.edu/COLOR/instances.html