Mandatory Assignment 2 - MGA

AI and IOT Student

by

Oskar Eftedal Markussen, Lars Erik Moi

group 30

in

IKT433-G 24V

Distributed and Big Data Systems

Guided by university lecturer Noureddine Bouhmala

Faculty of Engineering and Science

University of Agder

Grimstad, March 2024

Innhold

Figurer

1 Introduction

In this assignment given by the university we are tasked to colour a graph with two colours, black and white. This task, similar to the one shown in class, is to find the optimal pattern such that each colour is different from its connected vertices. In this graph we are meant to use 16 vertices and to choose the edges between the different vertices ourselves. These vertices are then coloured based on what is called the genetic algorithm, with a population of 4 individuals.

The given steps introduced to us by the assignment is first to find a random reduction method that reduces the problem from 16 vertices to 8 vertices and finally to 4 vertices. The second step that is introduced is to use the genetic algorithm for the problem with 4 vertices, and further explain the results of using two generations, as an example. This is done with a two-point crossover, and since it is a genetic algorithm we have decided to use mutation in our implementation. The third step introduced is to use the solution with the 4 vertices to solve the initial solution for the problem with 8 vertices. The fourth and last step was to use the solution from 8 vertices to solve the initial problem for 16 vertices. This then shows that the reduction of a problem could be utilized to simplify a optimization or problem.

The last part of this rapport will contain a short discussion on how we could implement our genetic algorithm differently and some difficulties we encountered during our implementation.

2 Theoretical background

2.1 Genetic algorithm

According to the paper ”Genetic Algorithm [1]“ by Tom V. Mathew, the Genetic Algorithms (GAs) are computational models inspired by natural evolution, leveraging mechanisms such as selection, crossover, and mutation to solve optimization problems. They operate on a population of potential solutions, encoded as chromosomes, to evolve increasingly fit solutions over generations. The process involves evaluating the fitness of each solution, selecting the fittest individuals for reproduction, and applying genetic operators to introduce variation. This population-based approach allows GAs to efficiently explore and exploit complex search spaces, making them versatile for a wide range of applications, from numerical optimization to machine learning and beyond. The success of GAs lies in their ability to balance exploration of the search space with the exploitation of high-fitness solutions, adapting to various problem domains without the need for gradient information or specific domain knowledge.

2.2 The colouring problem

According to the paper ”Genetic algorithm for black and white colouring problem on graphs [2]“ the colouring problem involves assigning colours to the vertices of a graph such that no two adjacent vertices share the same colour. In their study on the Black and White Colouring problem (BWC) within graph theory. The authors explore the effectiveness of genetic algorithms compared to the traditional Tabú Search method. They focus on a specialized case of the Graph Anti-Colouring Problem (GAC), aiming to maximize the number of vertices that do not neighbor black vertices. By employing genetic algorithms with parameters fine-tuned through the irace tool, they demonstrate that these algorithms can outperform or match the state-of-the-art solutions in most tested instances. This finding underscores the potential of genetic algorithms as a powerful and flexible tool for solving complex optimization problems in graph theory, offering a simpler implementation and comparable, if not superior, performance to existing methods.

3 Implementation

The following chapter explains our implementation of the assignment and how we have approached some of its issues.

3.1 Principal

The principle implementations we followed was to first initiate a set of 16 vertices and its corresponding edges. We choose a circular pattern as our static test, while also testing some random patterns produced by one of our functions. For demonstration purposes we will focus on the circular pattern of vertices and edges. After initialisation of the vertices we reduced the problem from 16 to 8 vertices by randomly making a set of vertices that where connected to each other. This was done ones more to reduce the problem domain down to 4 vertices as described in our assignment.

After reduction we evaluated fitness of the possible parents and then used a two-point crossover with mutation for a set amount of generations, in the case of our assignment it was 2 generations. After the crossover we check the fitness of both the parents and the children to find the current most optimal solution. Note this might not be a global maxima, but a local. With this we now have a solution for our 4 vertices problem which we use to initiate back to our 8 vertices problem. There are multiple ways of mapping the reduced 4 vertices solution to the full 8 vertices solution, however we decided to fill the unmapped vertices with Black ('B'). Initially filling the full solution with black ('B') serves the purpose of ensuring that vertices not included in the reduced solution (those that were not part of the optimization process) have a default value. This filled solution is then introduced to the next genetic algorithm with 8 vertices and this happens ones more for the 16 vertices problem.

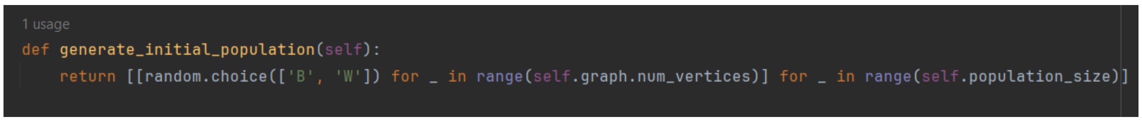
After the last genetic algorithm of the 16 vertices problem we have a optimized solution, to ensure a global maxima we would like to do more generations, and have a early stop when the fitness is fulfilled.

3.2 Explanation of code

In our code we uses three primary classes, which is a graph class, a visualize graph class and lastly a genetic algorithm class. The graph class is used to represent the graph information in this assignment such as the vertices and its edges. The visualize graph class is used as a way to represent our graph and solution visually.

Lastly is our genetic algorithm class which contain the functions generate_initial_population, fitness, select, two_point_crossover, mutate, run, reduce_graph, and expand_solution.

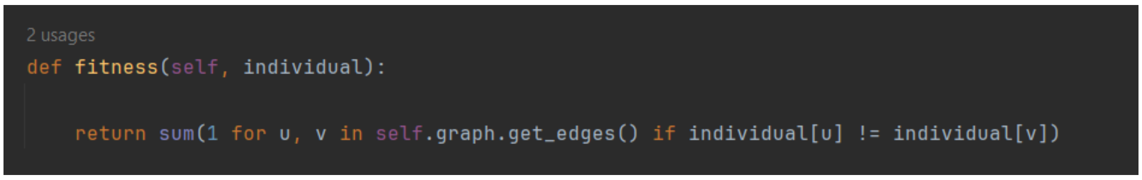
Generate_initial_population function generates an initial population for the genetic algorithm by creating a list of random 'B' (black) and 'W' (white) colour assignments for each vertex in the graph, doing so for a specified number of individuals (population size) as shown in figure 1.



```
1 usage
def generate_initial_population(self):
    return [[random.choice(['B', 'W']) for _ in range(self.graph.num_vertices)] for _ in range(self.population_size)]
```

Figur 1: Generate_initial_population

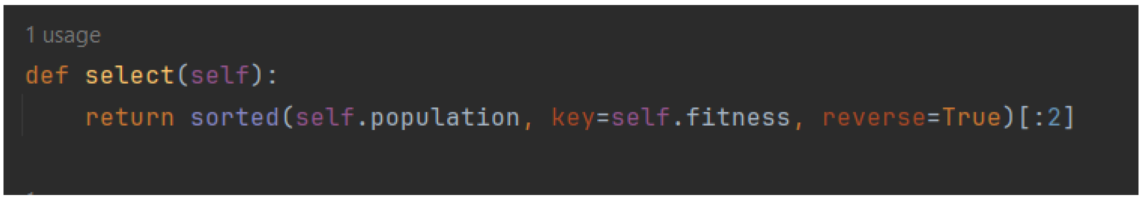
The fitness function calculates the fitness of an individual solution by counting the number of edges in the graph that connect vertices of different colours, as shown in figure 2



```
2 usages
def fitness(self, individual):

    return sum(1 for u, v in self.graph.get_edges() if individual[u] != individual[v])
```

Figur 2: fitness function

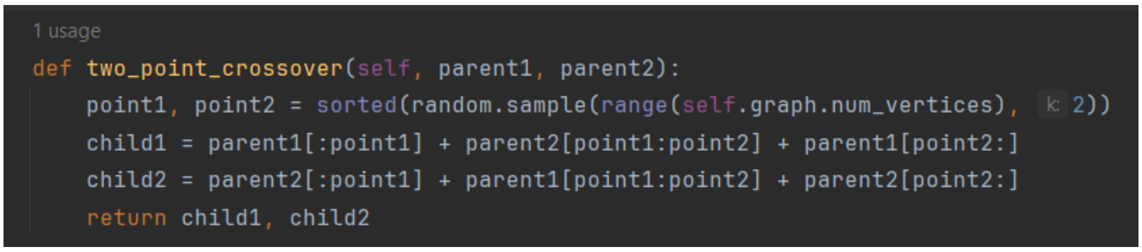
The select function selects the top two fittest individuals from the population by sorting all individuals in descending order based on their fitness (calculated via the fitness method) and then picking the first two from this sorted list, as shown in figure 3.



```
1 usage
def select(self):
    return sorted(self.population, key=self.fitness, reverse=True)[:2]
```

Figur 3: select function

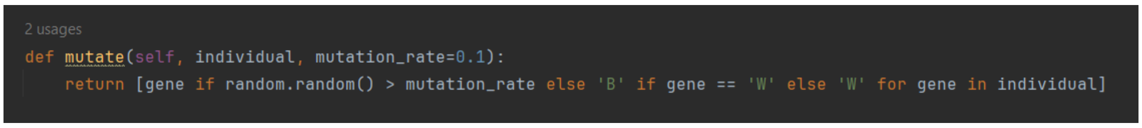
The two_point_crossover function performs a two-point crossover between two parent solutions by randomly selecting two points within the solutions and swapping the segments between these points to create two new child solutions as shown in figure 4.



```
1 usage
def two_point_crossover(self, parent1, parent2):
    point1, point2 = sorted(random.sample(range(self.graph.num_vertices), k: 2))
    child1 = parent1[:point1] + parent2[point1:point2] + parent1[point2:]
    child2 = parent2[:point1] + parent1[point1:point2] + parent2[point2:]
    return child1, child2
```

Figur 4: two-point crossover

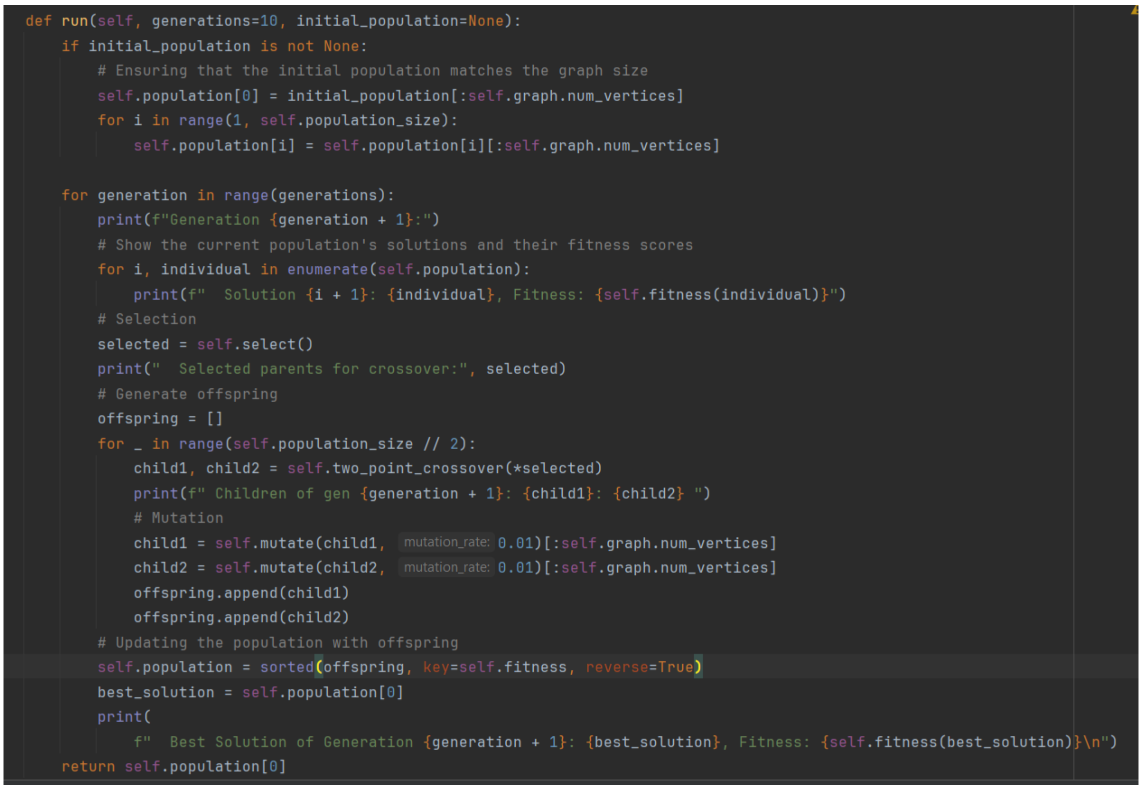
The mutate function mutates an individual solution by randomly flipping each gene (color) from ’B’ to ’W’ or vice versa with a specified probability (mutation_rate), as shown in figure 5.



```
2 usages
def mutate(self, individual, mutation_rate=0.1):
    return [gene if random.random() > mutation_rate else 'B' if gene == 'W' else 'W' for gene in individual]
```

Figur 5: mutate function

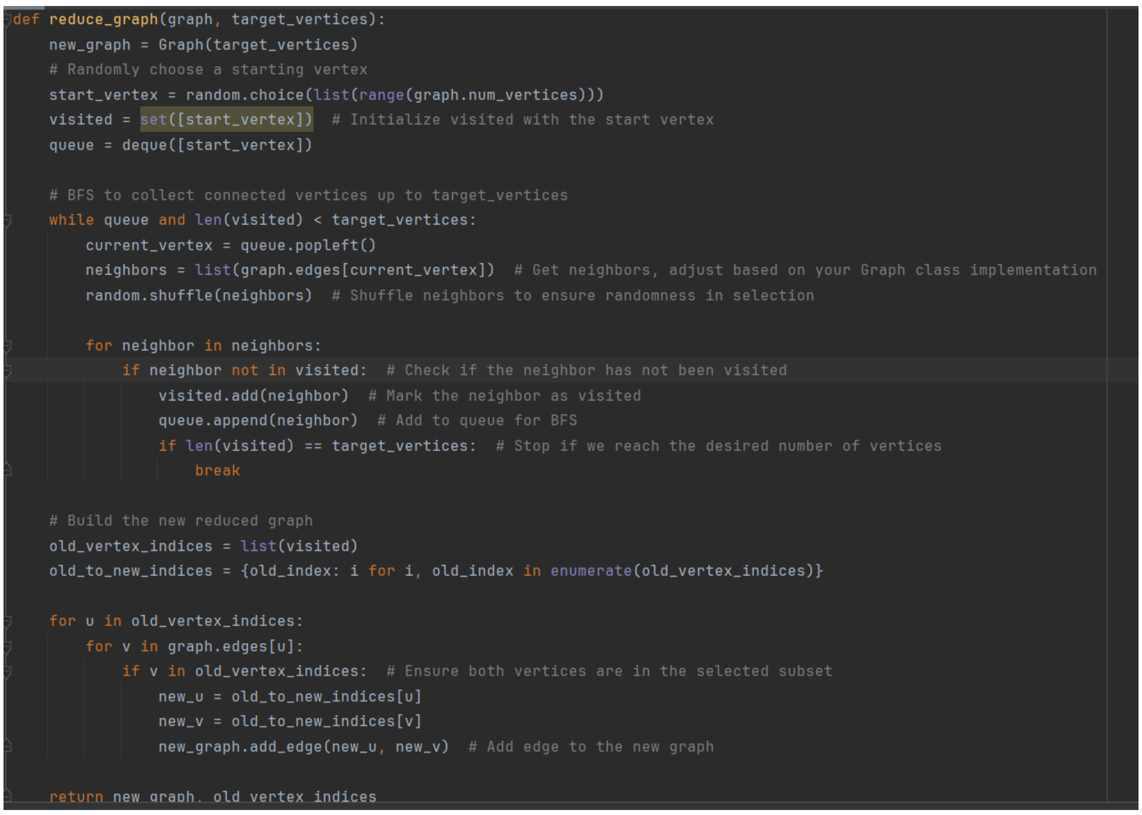
The run function executes the genetic algorithm over a specified number of generations, potentially starting with an initial population. It ensures the initial population fits the graph size, then iteratively selects the fittest individuals, generates offspring through crossover and mutation, and updates the population with these offspring, aiming to improve overall fitness. Finally, it returns the best solution found after all generations have been processed, as shown in figure 6.



```python
def run(self, generations=10, initial_population=None):
    if initial_population is not None:
        # Ensuring that the initial population matches the graph size
        self.population[0] = initial_population[:self.graph.num_vertices]
        for i in range(1, self.population_size):
            self.population[i] = self.population[i][:self.graph.num_vertices]

    for generation in range(generations):
        print(f"Generation {generation + 1}:")
        # Show the current population's solutions and their fitness scores
        for i, individual in enumerate(self.population):
            print(f"  Solution {i + 1}: {individual}, Fitness: {self.fitness(individual)}")
        # Selection
        selected = self.select()
        print("  Selected parents for crossover:", selected)
        # Generate offspring
        offspring = []
        for _ in range(self.population_size // 2):
            child1, child2 = self.two_point_crossover(*selected)
            print(f" Children of gen {generation + 1}: {child1}: {child2} ")
            # Mutation
            child1 = self.mutate(child1,  mutation_rate: 0.01)[:self.graph.num_vertices]
            child2 = self.mutate(child2,  mutation_rate: 0.01)[:self.graph.num_vertices]
            offspring.append(child1)
            offspring.append(child2)
        # Updating the population with offspring
        self.population = sorted(offspring, key=self.fitness, reverse=True)
        best_solution = self.population[0]
        print(
            f"  Best Solution of Generation {generation + 1}: {best_solution}, Fitness: {self.fitness(best_solution)}\n")
    return self.population[0]
```

Figur 6: genetic algorithm function

The reduce_graph function reduces a larger graph to a smaller graph with a specified number of vertices by randomly selecting a subset of vertices and then remapping and retaining the connected edges that exist between these selected vertices, effectively creating a simplified version of the original graph, as shown in figure 7.

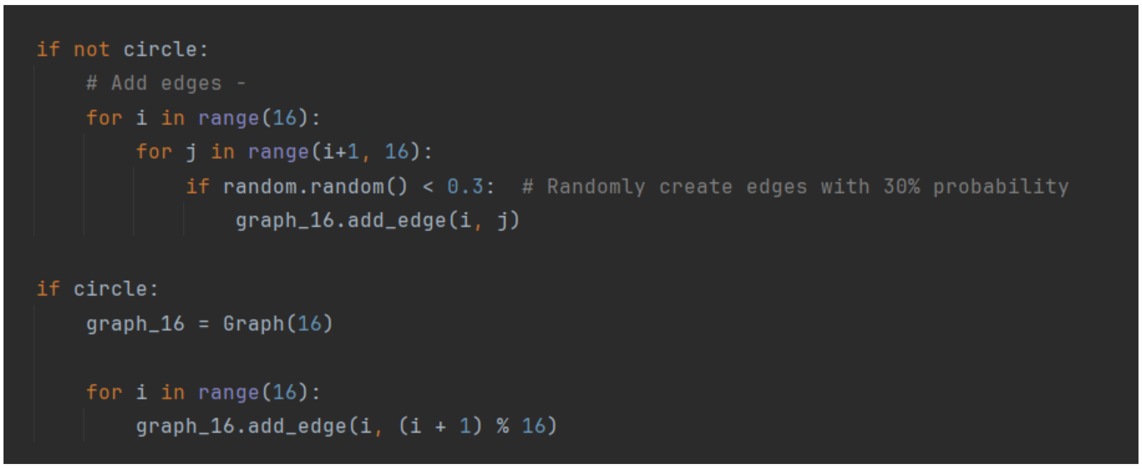


```python
def reduce_graph(graph, target_vertices):
    new_graph = Graph(target_vertices)
    # Randomly choose a starting vertex
    start_vertex = random.choice(list(range(graph.num_vertices)))
    visited = set([start_vertex])  # Initialize visited with the start vertex
    queue = deque([start_vertex])

    # BFS to collect connected vertices up to target_vertices
    while queue and len(visited) < target_vertices:
        current_vertex = queue.popleft()
        neighbors = list(graph.edges[current_vertex])  # Get neighbors, adjust based on your Graph class implementation
        random.shuffle(neighbors)  # Shuffle neighbors to ensure randomness in selection

        for neighbor in neighbors:
            if neighbor not in visited:  # Check if the neighbor has not been visited
                visited.add(neighbor)  # Mark the neighbor as visited
                queue.append(neighbor)  # Add to queue for BFS
                if len(visited) == target_vertices:  # Stop if we reach the desired number of vertices
                    break

    # Build the new reduced graph
    old_vertex_indices = list(visited)
    old_to_new_indices = {old_index: i for i, old_index in enumerate(old_vertex_indices)}

    for u in old_vertex_indices:
        for v in graph.edges[u]:
            if v in old_vertex_indices:  # Ensure both vertices are in the selected subset
                new_u = old_to_new_indices[u]
                new_v = old_to_new_indices[v]
                new_graph.add_edge(new_u, new_v)  # Add edge to the new graph

    return new_graph, old_vertex_indices
```

Figur 7: reduce graph function

The expand_solution function expands a solution from a reduced graph back to the original graph's size by initializing a solution array with a default colour ('B') for all vertices, and then updates this array with the colours from the reduced solution, mapping these colours to their corresponding vertices in the original graph based on previously selected indices.

Before initialization of the population we create the patterns that connects the vertices, by either adding in a random edge pattern or a general circular pattern, as shown in the figure 8



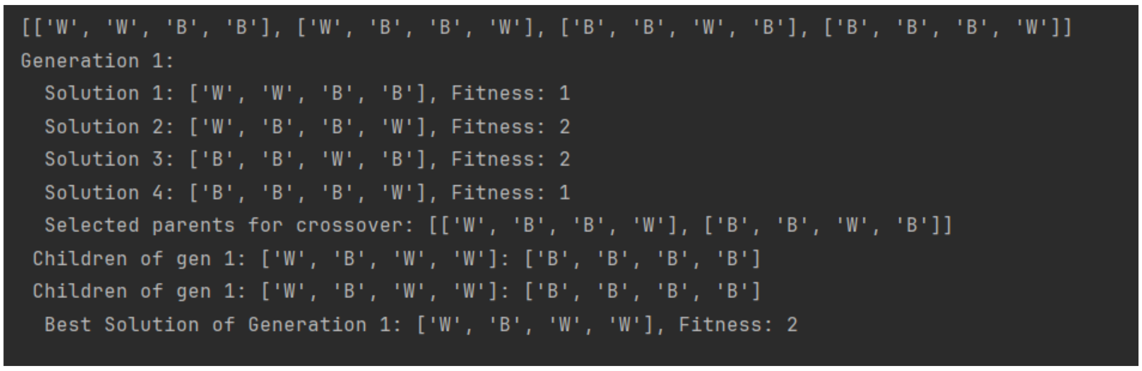
```python
if not circle:
    # Add edges -
    for i in range(16):
        for j in range(i+1, 16):
            if random.random() < 0.3:  # Randomly create edges with 30% probability
                graph_16.add_edge(i, j)

if circle:
    graph_16 = Graph(16)

    for i in range(16):
        graph_16.add_edge(i, (i + 1) % 16)
```

Figur 8: create edge patterns

These functions together is then used to optimize the coloring problem.

3.3 Result of optimization

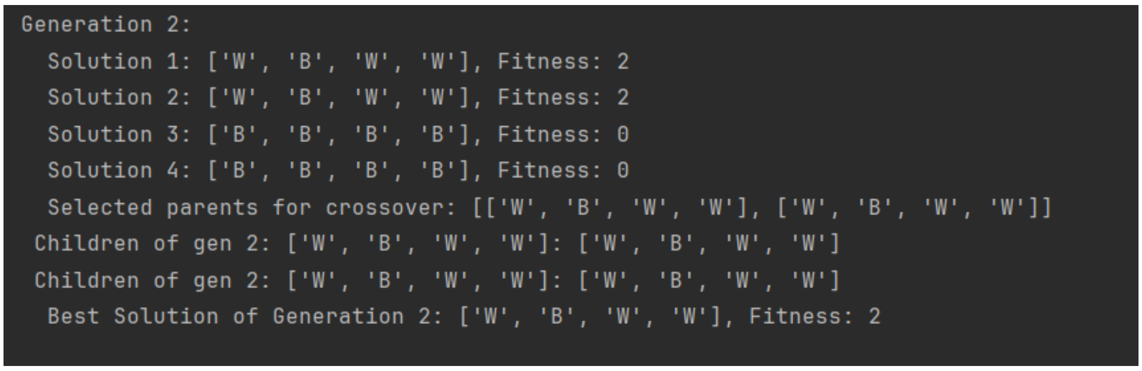
The results of our optimization was promising, to demonstrate as said in the assignment we will be using an example of two generations to explain the process. The first part of our results was the first generation from the 4 vertices problem which resulted in the output shown in figure 9. The best solution gave us a fitness of 2 which indicates a degree of 2 different edges in our subset. Which just means that we had two correct edges of the black and white colouring.



```
[['W', 'W', 'B', 'B'], ['W', 'B', 'B', 'W'], ['B', 'B', 'W', 'B'], ['B', 'B', 'B', 'W']]
Generation 1:
  Solution 1: ['W', 'W', 'B', 'B'], Fitness: 1
  Solution 2: ['W', 'B', 'B', 'W'], Fitness: 2
  Solution 3: ['B', 'B', 'W', 'B'], Fitness: 2
  Solution 4: ['B', 'B', 'B', 'W'], Fitness: 1
  Selected parents for crossover: [['W', 'B', 'B', 'W'], ['B', 'B', 'W', 'B']]
 Children of gen 1: ['W', 'B', 'W', 'W']: ['B', 'B', 'B', 'B']
 Children of gen 1: ['W', 'B', 'W', 'W']: ['B', 'B', 'B', 'B']
  Best Solution of Generation 1: ['W', 'B', 'W', 'W'], Fitness: 2
```

Figur 9: 4 vertices problem first generation

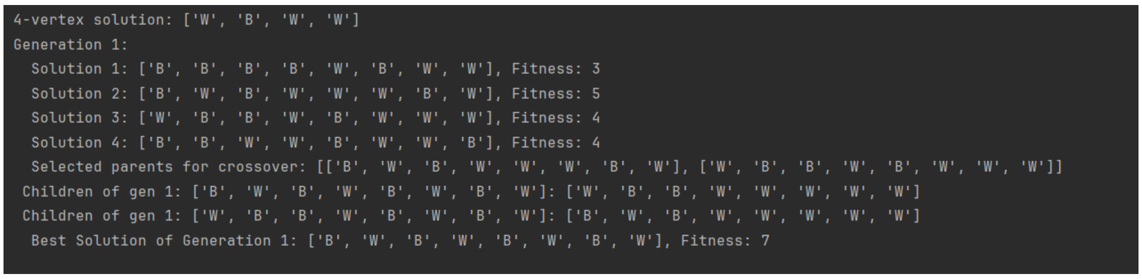
Following this we introduced the new child as a possible parent and started our second generation which resulted in the following result shown in figure 10. The result shows a fitness of two as well, much like our first generation.



```
Generation 2:
  Solution 1: ['W', 'B', 'W', 'W'], Fitness: 2
  Solution 2: ['W', 'B', 'W', 'W'], Fitness: 2
  Solution 3: ['B', 'B', 'B', 'B'], Fitness: 0
  Solution 4: ['B', 'B', 'B', 'B'], Fitness: 0
  Selected parents for crossover: [['W', 'B', 'W', 'W'], ['W', 'B', 'W', 'W']]
 Children of gen 2: ['W', 'B', 'W', 'W']: ['W', 'B', 'W', 'W']
 Children of gen 2: ['W', 'B', 'W', 'W']: ['W', 'B', 'W', 'W']
  Best Solution of Generation 2: ['W', 'B', 'W', 'W'], Fitness: 2
```

Figur 10: 4 vertices problem second generation

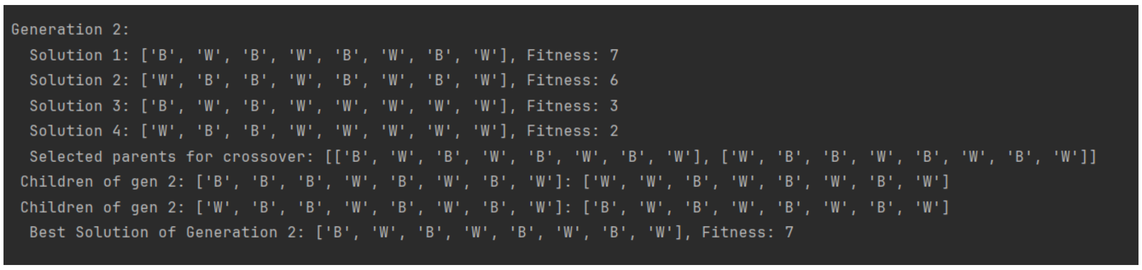
After finding a solution to the 4 vertices problem with two generations the process moves further on to the 8 vertices problem, where it introduces a expanded solution from the 4 vertices problem as a possible parent to the first generation. The result of the first generation is shown below in figure 11. As we can see in the figure, the past 4 vertices solution was not chosen as a parent for this generation, but the fitness from the child seemed promising as it is shown to be 7.



```
4-vertex solution: ['W', 'B', 'W', 'W']
Generation 1:
  Solution 1: ['B', 'B', 'B', 'B', 'W', 'B', 'W', 'W'], Fitness: 3
  Solution 2: ['B', 'W', 'B', 'W', 'W', 'W', 'B', 'W'], Fitness: 5
  Solution 3: ['W', 'B', 'B', 'W', 'B', 'W', 'W', 'W'], Fitness: 4
  Solution 4: ['B', 'B', 'W', 'W', 'B', 'W', 'W', 'B'], Fitness: 4
  Selected parents for crossover: [['B', 'W', 'B', 'W', 'W', 'W', 'B', 'W'], ['W', 'B', 'B', 'W', 'B', 'W', 'W', 'W']]
 Children of gen 1: ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']: ['W', 'B', 'B', 'W', 'W', 'W', 'W', 'W']
 Children of gen 1: ['W', 'B', 'B', 'W', 'B', 'W', 'B', 'W']: ['B', 'W', 'B', 'W', 'W', 'W', 'W', 'W']
  Best Solution of Generation 1: ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W'], Fitness: 7
```

Figur 11: 8 vertices problem first generation

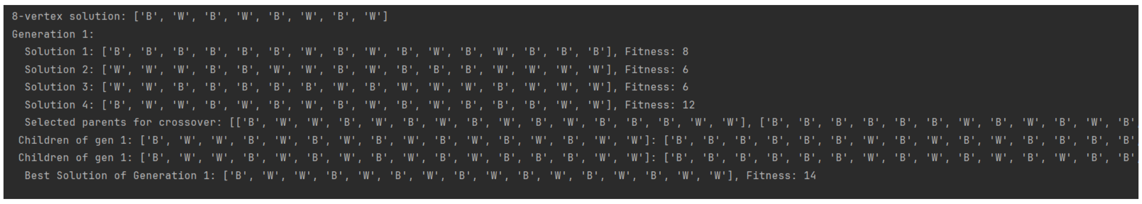
Further on in the algorithm the child with fitness 7 is introduced as a possible parent to the next generation, and we can see that it has been selected as a parent in our second generation shown in figure 12. However it did not yield any better result than its parents and the child from the first generation is chosen as the solution for the 8 vertices problem.



```
Generation 2:
  Solution 1: ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W'], Fitness: 7
  Solution 2: ['W', 'B', 'B', 'W', 'B', 'W', 'B', 'W'], Fitness: 6
  Solution 3: ['B', 'W', 'B', 'W', 'W', 'W', 'W', 'W'], Fitness: 3
  Solution 4: ['W', 'B', 'B', 'W', 'W', 'W', 'W', 'W'], Fitness: 2
  Selected parents for crossover: [['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W'], ['W', 'B', 'B', 'W', 'B', 'W', 'B', 'W']]
 Children of gen 2: ['B', 'B', 'B', 'W', 'B', 'W', 'B', 'W']: ['W', 'W', 'B', 'W', 'B', 'W', 'B', 'W']
 Children of gen 2: ['W', 'B', 'B', 'W', 'B', 'W', 'B', 'W']: ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']
  Best Solution of Generation 2: ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W'], Fitness: 7
```

Figur 12: 8 vertices problem second generation

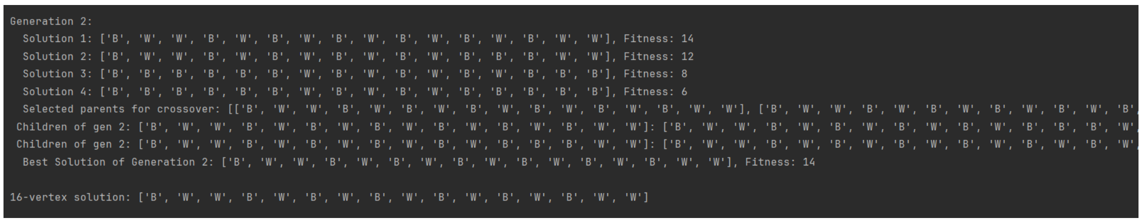
The expanded solution from the 8 vertices problem is now finally introduced into the original 16 vertices problem as a potential parent. Following this introduction the process is still the same as the previous genetic algorithms and the result from our first generation is shown in figure 13. We can here see that the previous solution is chosen as a parent together with a high fitness individual that have 12 in fitness. They together make a child that results in 14 total fitness.



```
8-vertex solution: ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']
Generation 1:
  Solution 1: ['B', 'B', 'B', 'B', 'B', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B', 'B'], Fitness: 8
  Solution 2: ['W', 'W', 'W', 'B', 'B', 'W', 'W', 'B', 'W', 'B', 'B', 'B', 'W', 'W', 'W', 'W'], Fitness: 6
  Solution 3: ['W', 'W', 'B', 'B', 'B', 'B', 'B', 'W', 'B', 'W', 'W', 'W', 'B', 'W', 'W', 'W'], Fitness: 6
  Solution 4: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B', 'B', 'W', 'W'], Fitness: 12
  Selected parents for crossover: [['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B', 'B', 'W', 'W'], ['B', 'B', 'B', 'B', 'B', 'B', 'W', 'B', 'W', 'B', 'W', 'B'
 Children of gen 1: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'W']: ['B', 'B', 'B', 'B', 'B', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B', 'B', 'B'
 Children of gen 1: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B', 'B', 'W', 'W']: ['B', 'B', 'B', 'B', 'B', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B'
  Best Solution of Generation 1: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'W'], Fitness: 14
```

Figur 13: 16 vertices problem first generation

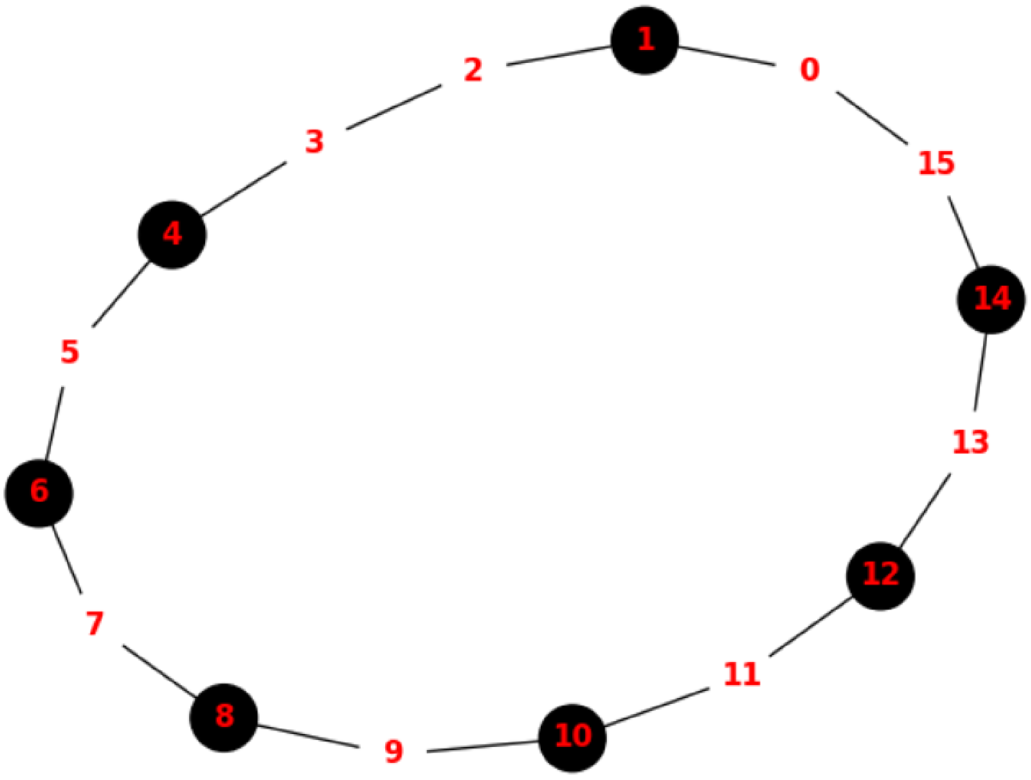
This child with the fitness 14 is then introduced as a parent to the second generation, and the result is shown below in figure 14. The child did not introduce a better fitness result so the final solution to the 16 vertices problem with two generations become a fitness of 14.



```
Generation 2:
  Solution 1: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'W'], Fitness: 14
  Solution 2: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B', 'B', 'W', 'W'], Fitness: 12
  Solution 3: ['B', 'B', 'B', 'B', 'B', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B', 'B'], Fitness: 8
  Solution 4: ['B', 'B', 'B', 'B', 'B', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B', 'B', 'B', 'B'], Fitness: 6
  Selected parents for crossover: [['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'W'], ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B'
 Children of gen 2: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'W']: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B', 'B', 'W'
 Children of gen 2: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'B', 'B', 'W', 'W']: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W'
  Best Solution of Generation 2: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'W'], Fitness: 14

16-vertex solution: ['B', 'W', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'W']
```

Figur 14: 16 vertices problem second generation

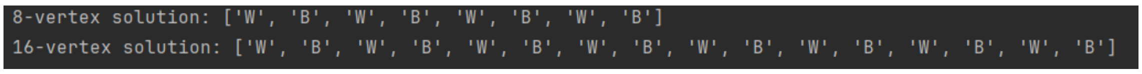
The visualisation of the final result from the 16 vertices is shown in figure 15



Figur 15: illustration of 16 vertices with 14 fitness

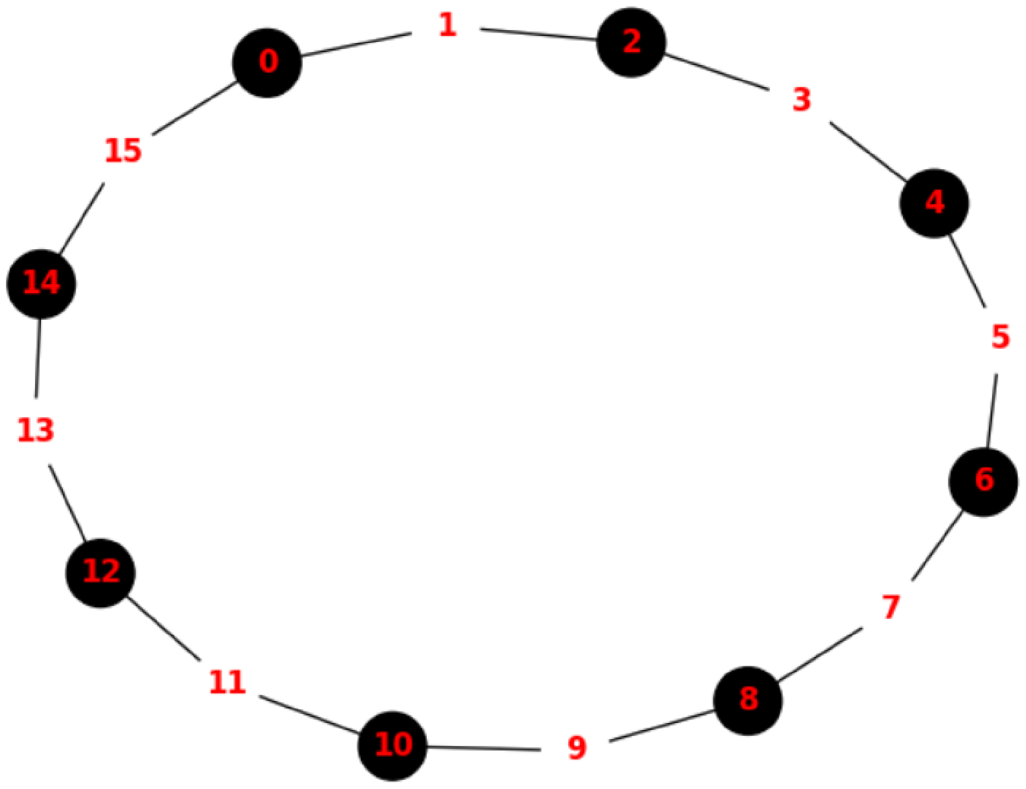
3.3.1 Addition of generations

Further to test our genetic algorithm we decided to add up to 100 generations in each run, giving us the most optimal solution to our problem which is shown in figure 16, and illustrated in figure 17.



```
8-vertex solution: ['W', 'B', 'W', 'B', 'W', 'B', 'W', 'B']
16-vertex solution: ['W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B']
```

Figur 16: 16 vertices with 100 generations



Figur 17: illustration of 16 vertices with 100 generations

This also shows that the usage of 2 generation was too little in our case to find the global maxima, but we found it with a 100 generations.

4 Discussion

4.1 Filling of the unmapped vertices

In our project we decided to expand our solution by appending the optimal solution from the previous vertices problem with all black vertices. This decision was done to demonstrate more of the algorithm it self rather than other strategies, however there is other strategies we could have used to improve our results. Examples of some strategies we could have used would be copy solution, two-way solution, random filling with proportional distribution, and nearest neighbor pattern.

The copy solution strategy would be to add the the solution to itself ones more, this would be if we expected a repeating pattern to occur as our solutions. The two-way solution would be to take the two with most fitness among the solutions to be added together, this would in our case make a better solution instead of our black filling's it would shorten the amount of generations needed to optimise. A random filling with proportional distribution would be to introduce a random filling of the unmapped vertices with a set distribution. This could be based on the distributions of black and white in the solution to be filled or some other distribution. The last strategy we considered was the nearest neighbor pattern, which would be to look at a end set of the solution to be expanded and then copy its patterns. This method could also be used if you expect there to be a repetitive based pattern as an optimised solution. However despite this possible strategies we chose to take the simplest one which was filling with a uniform color, black.

4.2 Estimated amount of generations needed

In our implementation chapter we demonstrated the result with two generation, this showed to be a good optimisation, but not a global maxima such as our 100 generation result, which had the maximum amount of fitness it could have. This raises the question how many generations would you need to reach a global maxima and there is no definitive answer, but we observed that the more generations that was added the better result we got. Following this we also observed that it would often be optimised with 50 generations, but with fewer it would often struggle to reach the maximum fitness for the model.

5 Conclusion

As a conclusion we have created a program that uses genetic algorithms to find an effective way to colour graphs, making it easier to solve big graph problems. Our program starts with a bunch of random solutions, picks the best ones, and mixes them, using two-point crossover, to get better solutions over generations. We improved it by breaking down big graphs into smaller pieces, solving the smaller problem first, and then applying that solution to the big graph again. This way, we don't have to work as hard on the big problem all at once. We also thought about different ways to decide on the colours for expanding our solution to the initial problem domain. Overall, our program is a good optimisation algorithm for tackling the tricky task of graph colouring, showing how genetic algorithms can be used to get optimised answers to semi tough problems.

Referanser

[1] Tom V Mathew. «Genetic algorithm». I: *Report submitted at IIT Bombay* 53 (2012).

[2] Luis Eduardo Urbán Rivero, Javier Ramírez Rodríguez og Rafael López Bracho. «Genetic algorithm for black and white coloring problem on graphs». I: *Programación matemática y software* 10.1 (2018), s. 17–23.

6 Code used:

Listing 1: Python code for GA

```python
import multiprocessing
import random
import itertools
from multiprocessing import Pool


from collections import deque
import matplotlib.pyplot as plt
import networkx as nx




def visualize_graph(graph, solution):
    G = nx.Graph()
    G.add_edges_from(graph.get_edges())

    # Add nodes explicitly based on the solution list
    G.add_nodes_from(range(1, len(solution) + 1))

    # Check if there's an extra node and remove it
    if len(G.nodes()) > len(solution):
        # Identify the extra node. Assuming it's the highest-
           numbered node
        extra_node = max(G.nodes())
        # Remove the extra node
        G.remove_node(extra_node)

    # Adjust color_map creation to account for the solution list
    color_map = ['black' if solution[node - 1] == 'B' else '
       white' for node in G.nodes()]

    # Draw the graph
    nx.draw(G, with_labels=True, node_color=color_map, node_size
       =700, font_weight='bold', font_color='red')
    plt.show()


```

```python
class Graph:
    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.edges = {i: set() for i in range(num_vertices)}

    def add_edge(self, u, v):
        if u != v and u < self.num_vertices and v < self.num_vertices:
             self.edges[u].add(v)
             self.edges[v].add(u)

    def get_edges(self):
        return [(u, v) for u in range(self.num_vertices) for v in self.edges[u] if u < v]




class GeneticAlgorithm:
    def __init__(self, graph, population_size=4):
        self.graph = graph
        self.population_size = population_size
        self.population = self.generate_initial_population()

    def generate_initial_population(self):
        return [[random.choice(['B', 'W']) for _ in range(self.graph.num_vertices)] for _ in range(self.population_size)]

    def fitness(self, individual):

        return sum(1 for u, v in self.graph.get_edges() if individual[u] != individual[v])

    def select(self):
        return sorted(self.population, key=self.fitness, reverse=True)[:2]

    def two_point_crossover(self, parent1, parent2):
        point1, point2 = sorted(random.sample(range(self.graph.num_vertices), 2))
        child1 = parent1[:point1] + parent2[point1:point2] + parent1[point2:]
        child2 = parent2[:point1] + parent1[point1:point2] + parent2[point2:]
        return child1, child2

    def mutate(self, individual, mutation_rate=0.1):
        return [gene if random.random() > mutation_rate else 'B' if gene == 'W' else 'W' for gene in individual]


    def run(self, generations=10, initial_population=None):
        if initial_population is not None:
            # Ensuring that the initial population matches the graph size
            self.population[0] = initial_population[:self.graph.num_vertices]
            for i in range(1, self.population_size):
                self.population[i] = self.population[i][:self.graph.num_vertices]

        for generation in range(generations):
            #print(f"Generation {generation + 1}:")
            # Show the current population's solutions and their fitness scores
            #for i, individual in enumerate(self.population):
            #    print(f"  Solution {i + 1}: {individual}, Fitness: {self.fitness(individual)}")
            # Selection
            selected = self.select()
            #print("  Selected parents for crossover:", selected)
            # Generate offspring
            offspring = []
            for _ in range(self.population_size // 2):
                child1, child2 = self.two_point_crossover(*selected)
                #print(f" Children of gen {generation + 1}: {child1}: {child2} ")
                # Mutation
                child1 = self.mutate(child1, 0.01)[:self.graph.num_vertices]
                child2 = self.mutate(child2, 0.01)[:self.graph.num_vertices]
                offspring.append(child1)
                offspring.append(child2)
            # Updating the population with offspring
            self.population = sorted(offspring, key=self.fitness, reverse=True)
            best_solution = self.population[0]
            #print(f"  Best Solution of Generation {generation + 1}: {best_solution}, Fitness: {self.fitness(best_solution)}\n")
        return self.population[0]



def reduce_graph(graph, target_vertices):
    new_graph = Graph(target_vertices)
    # Randomly choose a starting vertex
    start_vertex = random.choice(list(range(graph.num_vertices)))
    visited = set([start_vertex])  # Initialize visited with the start vertex
    queue = deque([start_vertex])

    # BFS to collect connected vertices up to target_vertices
    while queue and len(visited) < target_vertices:
        current_vertex = queue.popleft()
        neighbors = list(graph.edges[current_vertex])  # Get neighbors, adjust based on your Graph class implementation
        random.shuffle(neighbors)  # Shuffle neighbors to ensure randomness in selection

        for neighbor in neighbors:
            if neighbor not in visited:  # Check if the neighbor has not been visited
                visited.add(neighbor)  # Mark the neighbor as visited
                queue.append(neighbor)  # Add to queue for BFS
                if len(visited) == target_vertices:  # Stop if we reach the desired number of vertices
                    break

    # Build the new reduced graph
    old_vertex_indices = list(visited)
    old_to_new_indices = {old_index: i for i, old_index in enumerate(old_vertex_indices)}

    for u in old_vertex_indices:
        for v in graph.edges[u]:
            if v in old_vertex_indices:  # Ensure both vertices are in the selected subset
                new_u = old_to_new_indices[u]
                new_v = old_to_new_indices[v]
                new_graph.add_edge(new_u, new_v)  # Add edge to the new graph

    return new_graph, old_vertex_indices


def expand_solution(reduced_solution, old_vertex_indices, num_vertices):
    full_solution = ['B'] * num_vertices
    for new_index, color in enumerate(reduced_solution):
        old_index = old_vertex_indices[new_index]
        full_solution[old_index] = color
    return full_solution

def main():
    circle = True
    # Create the initial 16-vertex graph and add edges
    graph_16 = Graph(16)


    if not circle:
        # Add edges -
        for i in range(16):
            for j in range(i+1, 16):
                if random.random() < 0.3:  # Randomly create edges with 30% probability
                    graph_16.add_edge(i, j)

    if circle:
        graph_16 = Graph(16)

        for i in range(16):
            graph_16.add_edge(i, (i + 1) % 16)


    # Reduce graph from 16 to 8 vertices
    graph_8, indices_16_to_8 = reduce_graph(graph_16, 8)
    print(graph_8.edges)

    # Further reduce graph from 8 to 4 vertices
    graph_4, indices_8_to_4 = reduce_graph(graph_8, 4)
    print(graph_4.edges)

    # Apply GA to 4-vertex graph
    ga_4 = GeneticAlgorithm(graph_4)

    print(ga_4.population)
    solution_4 = ga_4.run(generations=100)
    print("4-vertex solution:", solution_4)

    # Expand solution from 4 to 8 vertices
    solution_8_initial = expand_solution(solution_4, indices_8_to_4, 8)

    # Apply GA to 8-vertex graph with initial solution
    ga_8 = GeneticAlgorithm(graph_8)
    solution_8 = ga_8.run(generations=100, initial_population=solution_8_initial)
    print("8-vertex solution:", solution_8)

    # Expand solution from 8 to 16 vertices
    solution_16_initial = expand_solution(solution_8, indices_16_to_8, 16)

    # Apply GA to 16-vertex graph with initial solution
    ga_16 = GeneticAlgorithm(graph_16)
    solution_16 = ga_16.run(generations=100, initial_population=
       solution_16_initial)
    print("16-vertex solution:", solution_16)
    visualize_graph(graph_16, solution_16)


if __name__ == '__main__':
    main()
```