Laboratorium 2 - sprawozdanie

Wstęp do sztucznej inteligencji

Marcin Grabysz

03.11.2021

1 Treść zadania

Tematem drugich ćwiczeń są algorytmy genetyczne i ewolucyjne. Państwa zadaniem będzie zaimplementować klasyczny algorytm ewolucyjny bez krzyżowania, z selekcją turniejową i sukcesją generacyjną. W raporcie należałoby wskazać jak zmiana liczby osobników w populacji wpływa na jakość uzyskanych rozwiązań przy ograniczonym budżecie. Warto również opisać zachowanie algorytmu dla różnych rodzajów danych wejściowych oraz wpływ zmiany parametrów. Przykładowe zbiory danych i/lub ich generatory należy samemu skonstruować na potrzebę zadania.

Układamy grafik dla studentów na zdalnym nauczaniu. Mamy n=25 przedmiotów (wierzchołki w grafie) i uczelnia najchętniej zrobiłaby je tego samego dnia. Niestety, jeżeli jakiś student zapisany jest na dwa przedmioty, to nie mogą one odbyć się o tej samej godzinie (student reprezentuje krawędź w grafie). Szukamy sposobu jak zorganizować wszystkie zajęcia w jak najmniejszej liczbie okien czasowych - innymi słowy jak pokolorować wierzchołki w grafie, by te same kolory nie były połączone krawędzią. (graph coloring problem).

2 Opis implementacji

2.1 Wykorzystane narzędzia

Algorytmy i funkcje służace badaniu funkcji zaimplementowane są w języku Python w wersji 3.8.10. Wykorzystane zostały biblioteki `matplotlib 3.4.3`

oraz `networkx 2.6.3`.

2.2 Opis działania

2.2.1 Zdefiniowanie osobnika

Aby zrozumieć działanie algorytmu ewolucyjnego, należy najpierw określić czym jest podstawowa komórka tego procesu - osobnik. W przyjętej implementacji osobnikiem jest obiekt klasy `Individual`, która posiada dwa atrybuty: `core` oraz `fitness_rate`.

`core` (nazwijmy go numerycznym schematem kolorów) jest przechowującą liczby całkowite listą o długości n (gdzie n jest liczbą wierzchołków w badanym grafie). Każdy kolejny element listy symbolizuje kolor (dowolny, ale określony) odpowiadającego mu wierzchołka. Rozważmy przykładowo numeryczny schemat kolorów dla pewnego grafu o pięciu wierzchołkach:

$$[2, 3, 3, 7, 2]$$

Informacja, jaką niesie dany schemat, jest następująca: wierzchołki 0. oraz 4. mają ten sam kolor (np. czerwony), wierzchołki 1. oraz 2. mają taki sam kolor (ale inny niż 0. i 4., np. niebieski) i wierzchołek 3. ma kolor inny niż wszystkie pozostałe (np. buropisiaty).

Wartości przechowywane w liście muszą należeć do przedziału od 0 do $n - 1$.

`fitness_rate` jest oceną jakości danego osobnika przypisywaną mu w czasie wykonywania funkcji `fitness()`. Jeżeli osobnik jest poprawny, tj. żadne dwa połączone wierzchołki nie są tego samego koloru, oceną danego osobnika jest liczba użytych kolorów (równa liczbie różnych wartości w numerycznym schemacie kolorów). Z tego powodu lepszym osobnikiem jest ten, którego `fitness_rate` jest niższe. W przypadku gdy osobnik nie jest poprawny, jego ocena jest ustawiana na $n+1$ (jest więc gorszy od każdego osobnika poprawnego i zawsze będzie przegrywał w reprodukcji turniejowej; chyba że trafi na również niepoprawnego przeciwnika).

2.2.2 evolutionary_algorithm

Algorytm jest realizowany przez funkcję `evolutionary_algorithm`, przyjmującą następujące argumenty:

- `iterations` - liczba iteracji do wykonania
- `edges` - obiekt klasy `EdgeView` lub lista krotek reprezentujących krawędzie w badanym grafie - przykładowo krotka (2, 4) oznacza, że wierzchołki 2. oraz 4. połączone są krawędzią
- `population_size=20` - rozmiar populacji
- `individual_size=25` - rozmiar osobnika (równy ilości wierzchołków w grafie)
- `mutation_ind_prob=0.2` - prawdopodobieństwo mutacji danego osobnika
- `mutation_elem_prob=0.08` - prawdopodobieństwo mutacji danego chromosomu
- `seed=None` - ziarno dla generatora liczb losowych. Podanie tego argumentu spowoduje powtarzalność otrzymanych wyników, więc zalecane jest aby tego nie robić w sytuacji innej niż testowanie lub debuggowanie.

Algorytm tworzy pewną populację początkową z wygenerowanych losowo osobników i zapmiętuje najlepszego z nich. W czasie kolejnych iteracji populacja poddana jest reprodukcji turniejowej, mutacji i ponownemu przeszukaniu populacji w celu znalezienia najlepszego osobnika.

Funkcja zwraca najlepszego znalezionego osobnika i obiekt klasy `database`, który przechowuje najważniejsze informacje o danym wykonaniu algorytmu takie jak: parametry wywołania funkcji, krawędzie grafu (pozwalają odtworzyć graf w wizualizacji), czas wykonania, średnią wartość oceny osobników w populacji oraz atrybuty najlepszego osobnika. Niniejsze dane można od razu zapisać w pliku .json, wykorzystując metodę `write_to_json()`.

2.2.3 Reprodukcja

Zadanie reprodukcji turniejowej populacji wykonuje funkcja `reproduction`. Przyjęte zostało założenie o turniejach stałej wielkości równej 2.

2.2.4 Mutacja

Mutacja jest realizowana przez funkcję `mutation`. Do zmutowania wybierane są (z prawdopodobieństwem równym `mutation_ind_prob`) pewne osobniki z populacji. Następnie pewne elementy z atrybutu `core` wybranego osobnika są zamieniane na losowe liczby z przedziału $[0, n-1]$ (prawdopodobieństwo zmutowania danego elementu listy jest równe parametrowi `mutation_elem_prob`).

2.3 Instrukcja użytkowania

Ze względu na znaczną liczbę parametrów, nie ma możliwości wywołania algorytmu z konsoli. Do tego działania jest jednak przygotowany plik `main.py`. Wszystkie parametry grafu, algorytmu i nazwę pliku .json można wprowadzić w kodzie (do czego zachęcam czytającego niniejsze sprawozdanie). Zakomentowana część służyła mi do prowadzenia i zapisywania pomiarów po 25-krotnym wykonaniu algorytmu.

Aby wygenerować wizualizację grafu i wykres średniej oceny osobników w populacji od liczby iteracji należy wywołać odpowiednio moduły `grapher.py` oraz `plotter.py`, podając jako argumenty nazwę otrzymanej grafiki oraz ścieżkę do pliku .json z danymi do zilustrowania. Można to zrobić z konsoli:

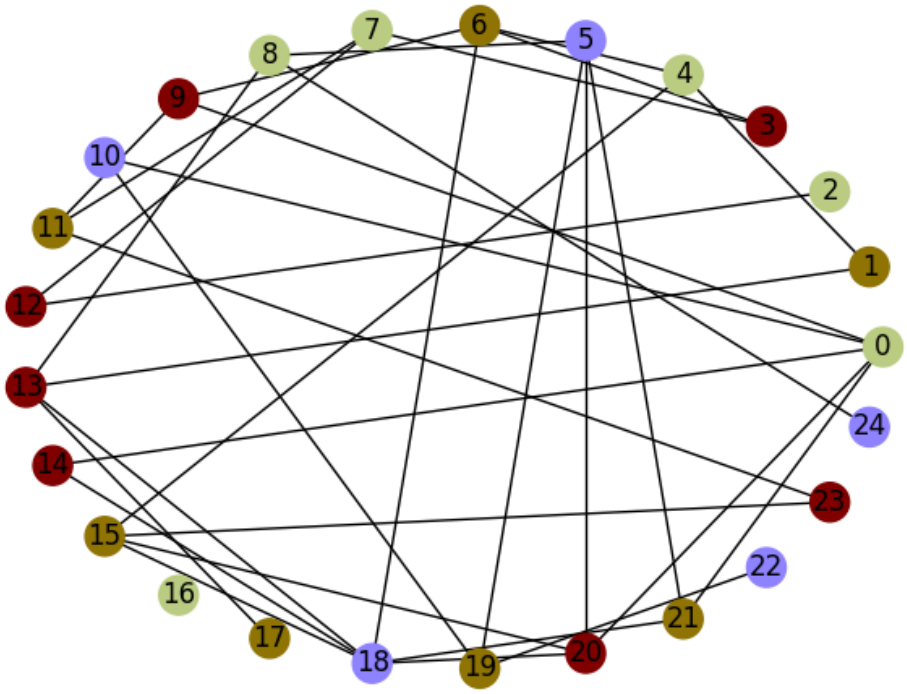
```
python3 grapher.py title path
```

3 Badanie implementacji

3.1 Badanie wstępne

W celu prezentacji działania algorytmu, zainicjowano utworzenie grafu o 25 wierzchołkach i prawdopodobieństwie utworzenia krawędzi równym 0.1. Graf jest powtarzalny i taki sam w następnych przykładach dzięki parametrowi seed = 1. Dla danego grafu uruchomiono algorytm ewolucyjny, przyjmując

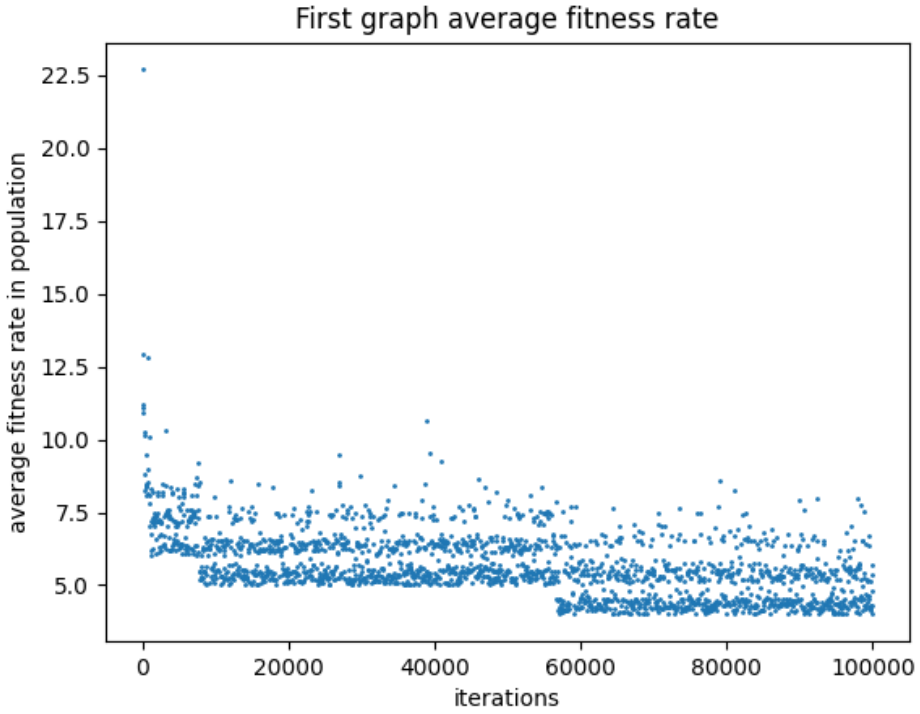
domyślne parametry i liczbę 100 000 iteracji. Po czasie 60,34 sekund otrzymano rozwiązanie, w którym do pokolorowania grafu użyto czterech różnych kolorów:



Rysunek 1: Wizualizacja pokolorowanego grafu

"Na oko" rozwiązanie wydaje się poprawne (ufam, że takie jest, skoro zostało pozytywnie ocenione przez funkcję `fitness()`). Być może nie jest *najlepsze*, ale na pewno jest bliskie najlepszemu - ponieważ jakakolwiek krawędź istnieje, wynik na pewno nie jest niższy niż 2. W przedziale wartości funkcji $[1, 26]$ wartość 4 należy uznać za dość dobrą.

Na podstawie otrzymanych danych można wygenerować także wiele mówiący wykres średniej oceny wszystkich osobników w populacji w zależności od liczby wkonanych iteracji (dla czytelności wykresu, uwzględniona jest co pięćdziesiąta iteracja):



Rysunek 2: Wykres jakości populacji od liczby iteracji

Daje się zauważyć, że jakość wygenerowanej losowo populacji początkowej jest bardzo słaba - prawdopodobnie wiele z osobników w ogóle nie spełnia założeń. Słabe osobniki są jednak szybko eliminowane w fazie reprodukcji i po niewielkiej liczbie iteracji jakość znacznie się polepsza, by osiągnąć średnią wartość równą 6, a potem 5. Okazuje się także, że warto było czekać całą minutę na wynik - po wykonaniu około 55 000 iteracji algorytm znalazł nowe minimum równe 4 i nie poprawił tego wyniku aż do końca działania.

3.2 Przeprowadzanie pomiarów

W dalszym badaniu dla konkretnego zestawu parametrów algorytm został wykonany 25 razy. Dla uproszczenia, ocenę najlepszego osobnika wygenerowanego przez jedno wykonanie algorytmu nazwijmy "wynikiem" (wynik jest równy najmniejszej znalezionej liczbie kolorów w grafie). W tabeli podane są:

- aktualnie badany parametr i jego wartość
- najlepszy wynik z 25 prób
- najgorszy wynik z 25 prób
- średni wynik z 25 prób
- odchylenie standardowe wyniku
- średni czas wykonania algorytmu w sekundach

Należy przyjąć, że pozostałe parametry wywołania funkcji są domyślne (patrz punkt: 2.2.2) a liczba iteracji równa 25 000. W każdym przypadku badany jest ten sam graf, zwizualizowany na rysunku 1.

parametr	min	max	średni w.	odch. st.	czas [s]
p_size=10	4	5	4,92	0,28	8,05
p_size=20	4	5	4,28	0,46	14.63
p_size=40	4	5	4,16	0,37	28,00
m_ind_prob=0,1	4	6	4,80	0,58	14,62
m_ind_prob=0,2	4	5	4,28	0,46	14,63
m_ind_prob=0,3	4	5	4,36	0,49	14,86
m_ind_prob=0,4	4	5	4,84	0,37	14,89
m_elem_prob=0,02	4	5	4,52	0,51	14,63
m_elem_prob=0,04	4	5	4,16	0,37	14,93
m_elem_prob=0,08	4	5	4,28	0,46	14,63
m_elem_prob=0,12	3	5	4,48	0,59	14,58
m_elem_prob=0,16	4	6	5,08	0,49	14,70

3.3 Wnioski

Wyraźne jest, że zwiększanie populacji przynosi korzyści (przynajmniej w badanym zakresie), ale wiąże się z liniowym przyrostem czasu działania (uzasadnione, bo reprodukcję, mutację i sprawdzenie jakości należy wykonać w każdej iteracji dla każdego osobnika).

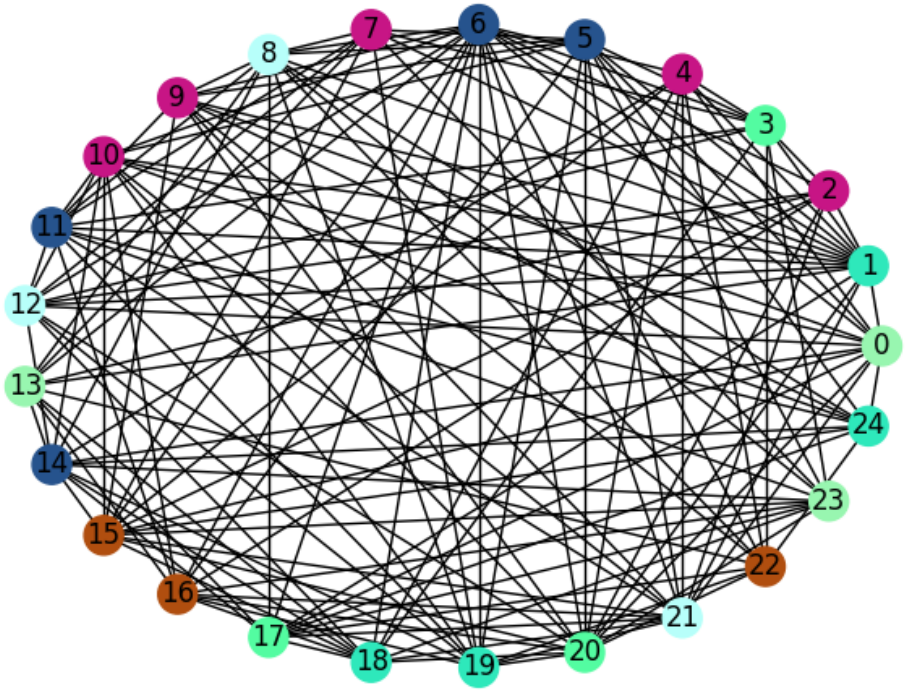
Co do obu parametrów prawdopodobieństwa mutacji, to wydaje się, że wartości domyślne są dość dobre (co ciekawe, bo przyjęte zostały w gruncie

rzeczy przypadkowo). Za parametr jakości uznaję tutaj średni wynik; wprawdzie dla prawdopodobieństwa mutacji elementu równego $0,12$ udało się uzyskać wynik 3 (a zatem najlepszy ze wszystkich), ale należy to traktować jako przypadek, co potwierdza również największe odchylenie standardowe sugerujące duże rozproszenie danych.

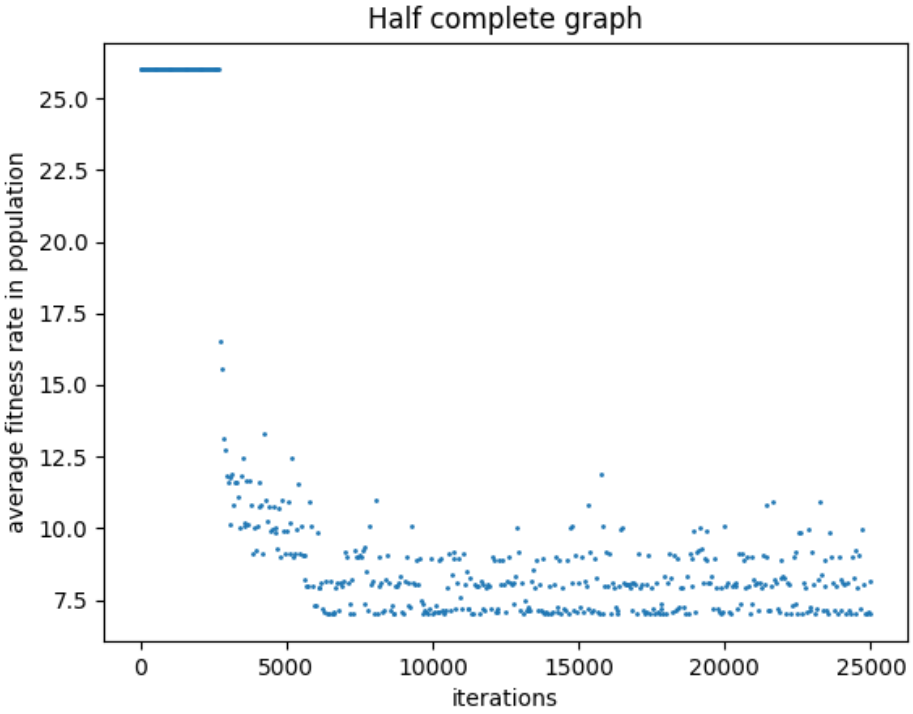
3.4 Inne grafy

Badania w tym rozdziale przeprowadzono, przyjmując parametry prawdopodobieństwa uznane za najlepsze w punkcie poprzednim (mutacji osobnika - 0,2; chromosomu - 0,04), rozmiar populacji równy 20 i liczbę iteracji 25 000 ze względu na czas wykonania. Każde badanie powtórzono 25 razy. Wizualizacje dotyczą ostatniego z dwudziestu pięciu przypadków.

3.4.1 Graf pełny z 50% usuniętych krawędzi



Rysunek 3: Wizualizacja pokolorowanego grafu



Rysunek 4: Średnia jakość populacji w kolejnych iteracjach

Zbiorcze wyniki zaprentowane są w tabeli:

min	max	średni w.	odch. st.	czas [s]
7	8	7,32	0,48	23,76

Czas wykonania dłuższy niż w poprzednich przypadkach wiąże się z tym, że funkcja `fitness()` iteruje po liście krawędzi danego grafu - w badanym grafie krawędzi jest znacznie więcej niż w poprzednim.

3.4.2 Graf pełny

Zbiorcze wyniki zaprezentowane są w tabeli:

min	max	średni w.	odch. st.	czas [s]
26	26	26	0	11,22

Analiza wyników prowadzi do gorzkiego wniosku, że algorytm nie znalazł rozwiązania dla grafu pełnego i co za tym idzie, nie jest całkowicie poprawny.

Ze względu na czas poświęcony na przeprowadzenie badań algorytmu, postanowiłem nie modyfikować danej implementacji. Jestem jednak świadomy (a przynajmniej przypuszczam) jakie założenia przyczyniły się do takiego błędu i jakie modyfikacje mogą go rozwiązać.

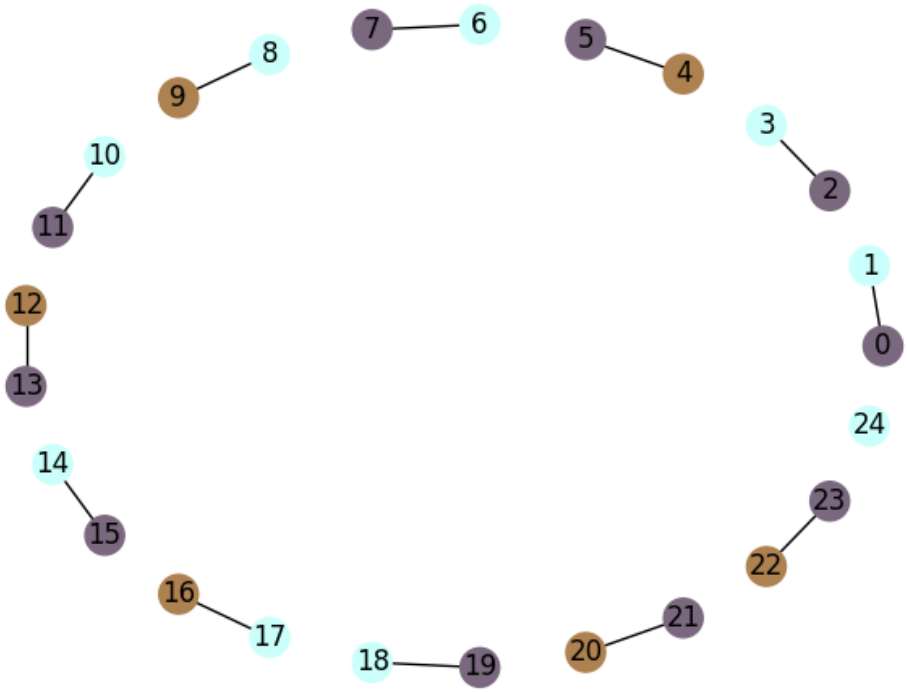
W danej implementacji nie ma rozróżnienia między osobnikami niepoprawnymi - każdy otrzymuje taką samą ocenę $n+1$. Oczywistym jest jednak, że w problemie kolorowania grafu pełnego, osobnik używający $n-1$ kolorów jest rozwiązaniem bliższym poprawnemu niż osobnik używający np. dwóch kolorów. W takim przypadku lepszy osobnik powinien być premiowany, gdyż istnieje duże prawdopodobieństwo, że w wyniku mutacji stanie się poprawny. W przypadku danej implementacji szukanie rozwiązania dla grafu pełnego polegało na oczekiwaniu, że przypadkiem pojawi się jedyny możliwy poprawny osobnik (czyli wykorzystujący dokładnie n kolorów), co oczywiście mogło się zdarzyć, ale w żadnym z dwudziestu pięciu wykonań się nie zdarzyło.

Wyjątkowo krótki czas wykonywania algorytmu wiążę się z tym, że po znalezieniu pary połączonych wierzchołków w tym samym kolorze funkcja `fitness()` przerywa dalsze przeszukiwanie zbioru krawędzi - w grafie pełnym zdarzało się nad wyraz często.

3.4.3 Graf dwudzielny

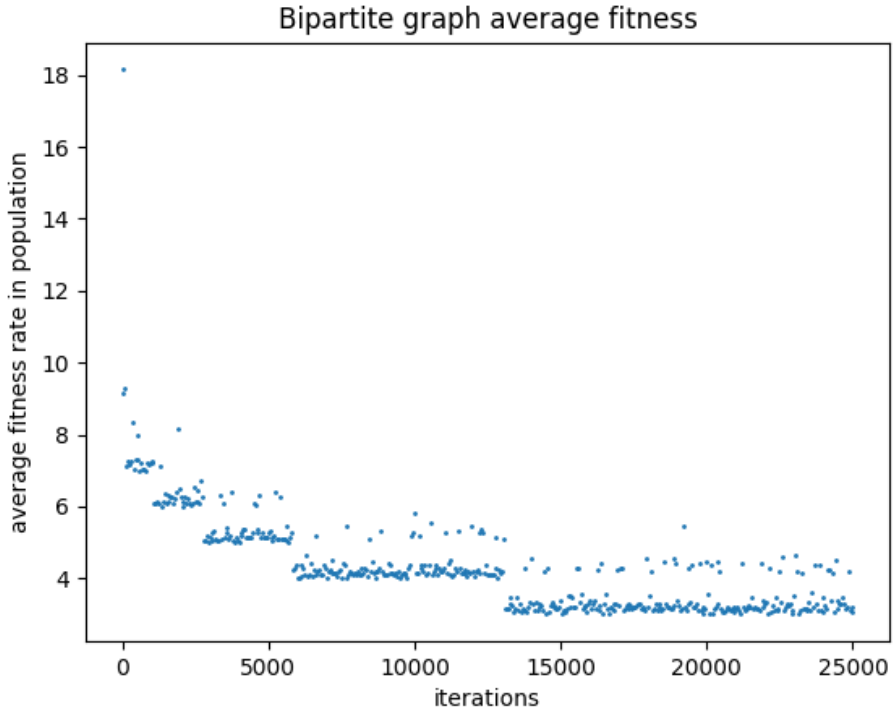
Zbiorcze wyniki zaprezentowane są w tabeli:

min	max	średni w.	odch. st.	czas [s]
3	5	3,76	0,52	9,74



Rysunek 5: Wizualizacja pokolorowanego grafu

Daje się zauważyć, że znalezione zostało poprawne (choć nie najlepsze) rozwiązanie.



Rysunek 6: Średnia jakość populacji w kolejnych iteracjach

4 Uwaga końcowa

Z własnej ciekawości postanowiłem dokonać modyfikacji algorytmu tak, aby działał skutecznie dla grafu o dużej liczbie krawędzi (w szczególności dla grafu pełnego). W nowej wersji funkcja `fitness()` sprawdza każdą krawędź i zachowuje liczbę niepoprawnych par p (takich, które są połączone i mają ten sam kolor). Ocena niepoprawnego osobnika równa się $n + p$. Ta drobna modyfikacja rozwiązuje problem, choć nieznacznie wydłuża działanie algorytmu.

Ponieważ problem został wykryty w momencie, w którym przeprowadzono już wiele badań i zgromadzono wiele danych, niniejsze sprawozdanie dotyczy pierwotnej wersji implementacji.