

Znajdowanie najkrótszej drogi w grafie ważonym(Python)

Marek Wiecaszek

April 26, 2022

1 Wstęp

Rozważać będziemy problem komiwojażera, czyli będziemy próbowali znaleźć drogę przez graf ważony o najmniejszym koszcie, która odwiedza każdy wierzchołek.

Rozwiązanie takiego problemu dla małych grafów jest trywialne, można bowiem, korzystając z klasycznych funkcji, niskim kosztem znaleźć wszystkie możliwe ścieżki po czym wybrać najlepszą. Jednak sprawa zaczyna się komplikować dla grafów o większej złożoności. Przykładowo dla grafu, który posiada 100 wierzchołków, a każdy z nich 3 połączenia, liczba możliwych dróg odwiedzających każdy wierzchołek (zakładając, że uda nam się to osiągnąć najmniejszą możliwą liczbą ruchów) będzie trochę mniejsza od 3^{100} . Jest to liczba z 29 zerami, a liczba wykonanych operacji dla każdej drogi to co najmniej 100n.

Można jednak spróbować rozwiązać ten problem algorytmami genetycznymi, który pozwoli nam znaleźć drogę bliską najlepszej dużo mniejszym kosztem. W moim rozwiązaniu jest to w $2^{\text{liczba wierzchołków}} \cdot \text{liczba generacji} \cdot \text{rozwiązania na generację} \cdot n$, gdzie n to stałe obliczenia wykonywane przy każdej iteracji.



2 Algrorytm i jak działa

2.1 Implementacja grafu

Aby móc zaimplementować graf stworzyłem dwie klasy obiektów klasa City zawierająca wierzchołki wraz z nazwa i potrzebna później funkcjonalnością i anologicznie klasę Route stanowiącą krawędzie.

```
class City:
    def __init__(self, name, connections):
        self.name = name
        self.connections = connections

    def add_connection(self, neighbour, route):
        self.connections.append(route)
        neighbour.connections.append(route)

    def delete_connection(self, route):
        self.connections.remove(route)

    def connected_with_indexes(self, l):
        result = []
        for con in self.connections:
            if con.connects[0] != self:
                result.append(l.index(con.connects[0]))
            else:
                result.append(l.index(con.connects[1]))
        return result

    def find_connection_with(self, other_city):
        for con in self.connections:
            if con.connects[0] == self and con.connects[1] == other_city or \
               con.connects[1] == self and con.connects[0] == other_city:
                return con
```

```
class Route:
    def __init__(self, name, cost, connects):
        self.cost = cost
        self.name = name
        self.connects = connects

    def change_cost(self, new_cost):
        self.cost = new_cost

    def show_route(self):
        print(self.connects[0].name, "---", self.cost, "---", self.connects[1].name)
```

Oraz funkcje odpowiedzialna za samo jego skonstruowanie

```
def make_map():
    def check_duplicate(c1, c2, routes):
        if c1 == c2:
            return False
        for r in routes:
            if r.connects[0] == c1 and r.connects[1] == c2 or r.connects[0] == c2 and r.connects[1] == c1:
                return False
        return True

    def make_cons():
        city_list = []
        file = open("cities.txt", "r")
        names = []
        for i in file:
            names.append(i.split(",")[0])
        for i in range(number_of_cities):
            city_list.append(City(names[random.randint(0, len(names) - 1)], []))
        number_of_routes = number_of_cities * 3
        max_cost = 250
        route_list = []
        i = 0
        while i < number_of_routes:
            city_1 = city_list[random.randint(0, number_of_cities - 1)]
            city_2 = city_list[random.randint(0, number_of_cities - 1)]
            if check_duplicate(city_1, city_2, route_list):
                con = Route(random.randint(0, len(names) - 1), random.randint(5, max_cost), [city_1, city_2])
                route_list.append(con)
                city_1.add_connection(city_2, con)
                i += 1
            return city_list, route_list
        bad_connections = True
        c, r = make_cons()
        while bad_connections:
            bad_connections = False
            for city in c:
                if len(city.connections) < 2:
                    bad_connections = True
                    c, r = make_cons()
                    print("Remaking the list")
                    break
        return c, r
```

2.2 Funkcja fitness

Algorytm genetyczny jak sama nazwa wskazuje działa na podstawie genetyki. Na początku tworzone są losowe rozwiązania z pośród, których wybierane są najlepsze. Następnie na podstawie określonej ilości najlepszych rozwiązań generowane jest kolejne pokolenie. Algorytm kończy swoje działanie po wykonaniu ustalonej ilości operacji lub osiągnięciu podanej wartości końcowej. Do oceniania wyników służy funkcja fitness (indywidualna dla każdego problemu) przedstawiona poniżej

```

def fitness_func(solution, solution_idx):
    cities_visited = [starting_point]
    previous_city = cities_1[starting_point]
    fitness = 0
    for move in solution:
        legal_moves = previous_city.connected_with_indexes(cities_1)
        if int(move) in legal_moves:
            con = cities_1[int(move)].find_connection_with(previous_city)
            fitness += con.cost
            previous_city = cities_1[int(move)]
            if int(move) not in cities_visited:
                cities_visited.append(int(move))
        else:
            fitness += 1000

    if len(cities_visited) == number_of_cities:
        fitness = fitness / 3
    print("operacja zakończona: ", solution_idx)

    return -int(fitness)

```

Poniżej znajduje się przykładowa konfiguracja wywołania algorytmu generycznego (wartości w zależności od potrzeb ulegają zmianie)

```

gene_space = list(range(0, number_of_cities))

sol_per_pop = 1000
num_genes = number_of_cities * 2

num_parents_mating = 150
num_generations = 5000
keep_parents = 100

parent_selection_type = "sss"

crossover_type = "single_point"

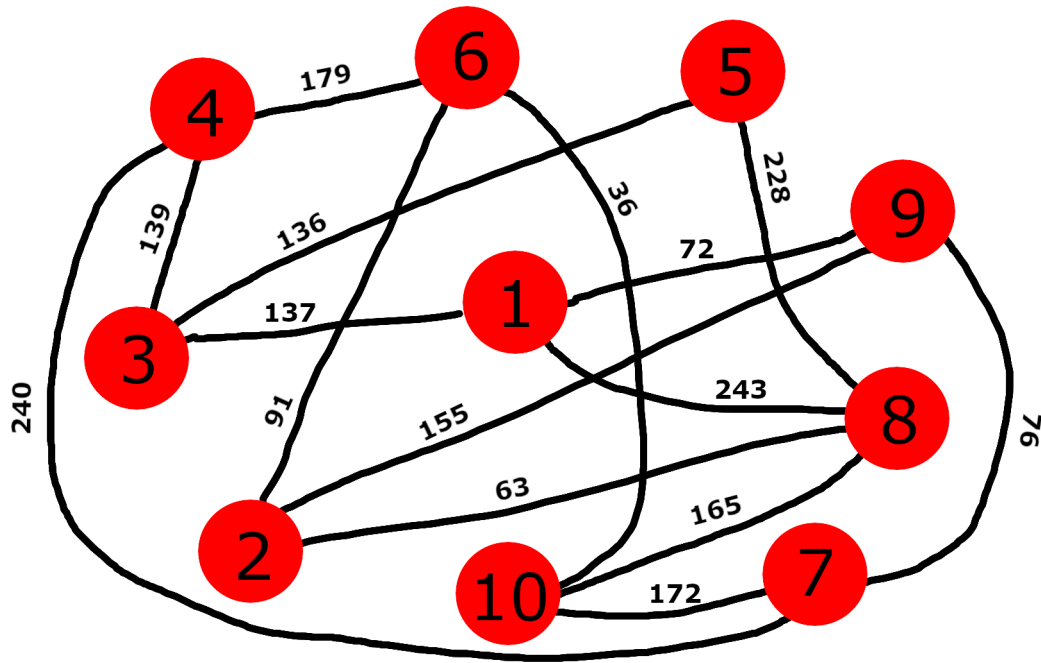
mutation_type = "random"
mutation_percent_genes = 100 / number_of_cities + 1

ga_instance = pygad.GA(gene_space=gene_space,
                        num_generations=num_generations,
                        num_parents_mating=num_parents_mating,
                        fitness_func=fitness_function,
                        sol_per_pop=sol_per_pop,
                        num_genes=num_genes,
                        parent_selection_type=parent_selection_type,
                        keep_parents=keep_parents,
                        crossover_type=crossover_type,
                        mutation_type=mutation_type,
                        mutation_percent_genes=mutation_percent_genes)

```

3 Przykładowe wywołanie

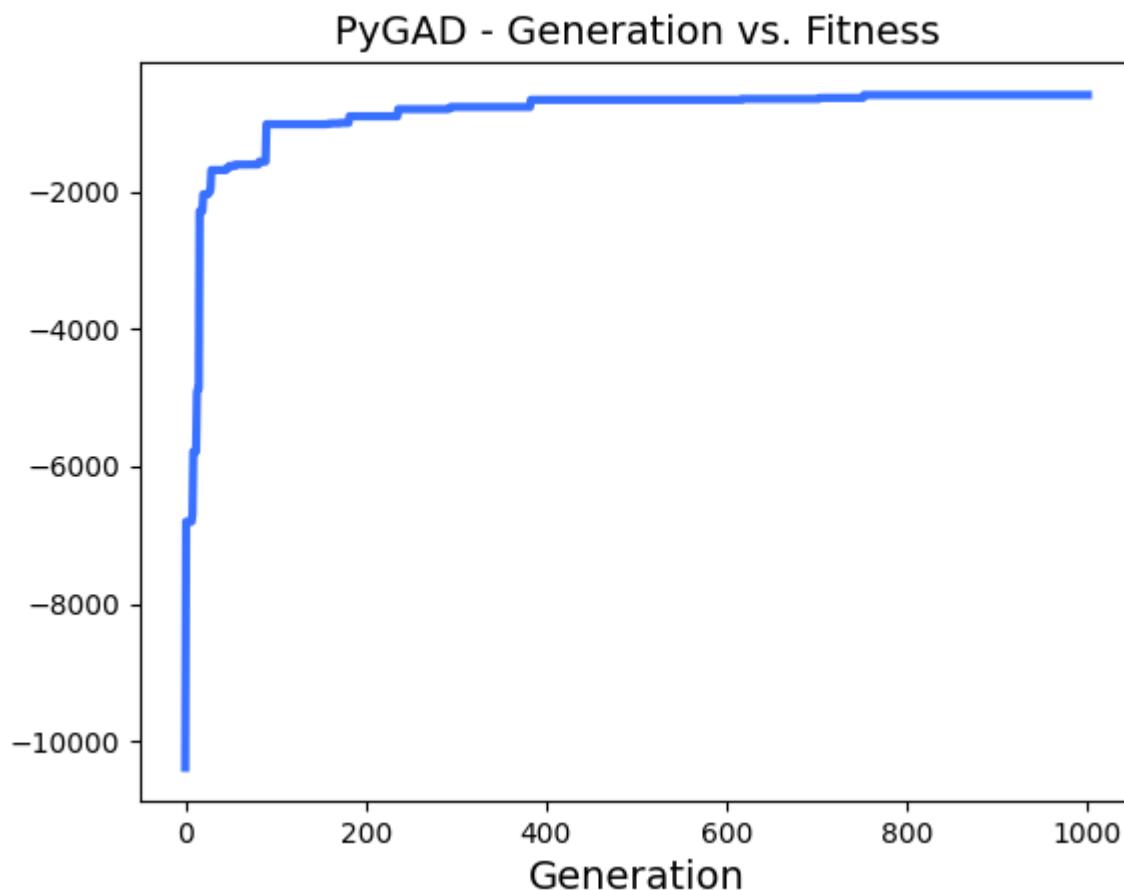
3.1 Dla przejrzystości najpierw zobaczmy jak działa algorytm dla grafów, które jesteśmy w stanie objąć wzorkiem



Jak widzimy powyższy graf nie jest bardzo skomplikowany i zapewne człowiek byłby w stanie znaleźć dla niego optymalne rozwiązanie w przeciągu kilku minut. Dla komputera to zadanie zajęło 7 sekund co nie wydaje się być mocno imponujące, trzeba jednak zwrócić uwagę, że aby ten problem rozwiązać trzeba myśleć z czym urządzenie nie radzi sobie zbyt dobrze. Dlatego ten wynik można uznać za dość dobry.

Droga, którą zwrócił algorytm to 1-9-7-10-6-2-8-5-3-4 i jej koszt wynosi 1013. Jeżeli ten wynik nie jest satysfakcjonujący polecam spróbować znaleźć lepsze rozwiązanie samemu (mi się nie udało)

3.2 Rozwój algorytmu



Wykres przedstawia wyniki zwracane przez funkcje fitness dla poszczególnych pokoleń. Jak widać różnią się one od ostatecznego wyniku 1013. Jak został uzyskany przedstawione jest poniżej.

3.3 Czyszczenie wyników

Oczywiście aby algorytm mógł się rozwijać potrzebna mu była większa liczba ruchów niż końcowa. Przyjąłem założenie, że $2 \times \text{liczba wierzchołków}$ będzie wystarczające. Oznacza to jednak, że na koniec będzie trzeba te wyniki oczyścić z niepotrzebnych ruchów. Do takich zaliczamy okazjonalne niepotrzebne zapętlenie ruchów czy tzw. ogon, czyli ruchy wykonane po odwiedzeniu wszystkich wierzchołków. Użyłem do tego funkcji `solution_cleanup`, która oprócz oczyszczania wyników prezentuje je w bardziej zrozumiały sposób.

```

def solution_cleanup(solution):
    real_weight = 0
    moves_used = [0]
    cities_visited = [starting_point]
    previous_city = cities_1[starting_point]
    two_back = cities_1[starting_point]
    for move in solution:
        if len(cities_visited) == number_of_cities:
            print("Odwiedzono wszystkie miasta")
            return real_weight, moves_used
        legal_moves = previous_city.connected_with_indexes(cities_1)
        if int(move) in legal_moves:
            moves_used.append(int(move))
            con = cities_1[int(move)].find_connection_with(previous_city)
            if two_back != cities_1[int(move)]:
                real_weight += con.cost
            two_back = previous_city
            previous_city = cities_1[int(move)]
            if int(move) not in cities_visited:
                cities_visited.append(int(move))
        else:
            print("Znaleziono nielegalny ruch (poleciał samolotem)")
    print("Odwiedzono miasta: ", len(cities_visited))
    return real_weight, moves_used

```

3.4 Inne rozwiązanie - metoda roju (swarm)

Innym sposobem rozwiązania tego problemu może być algorytm roju. Przedstawie go jednak bardziej jako ciekawostkę, gdyż po licznych testach okazał się być gorszym wyborem.

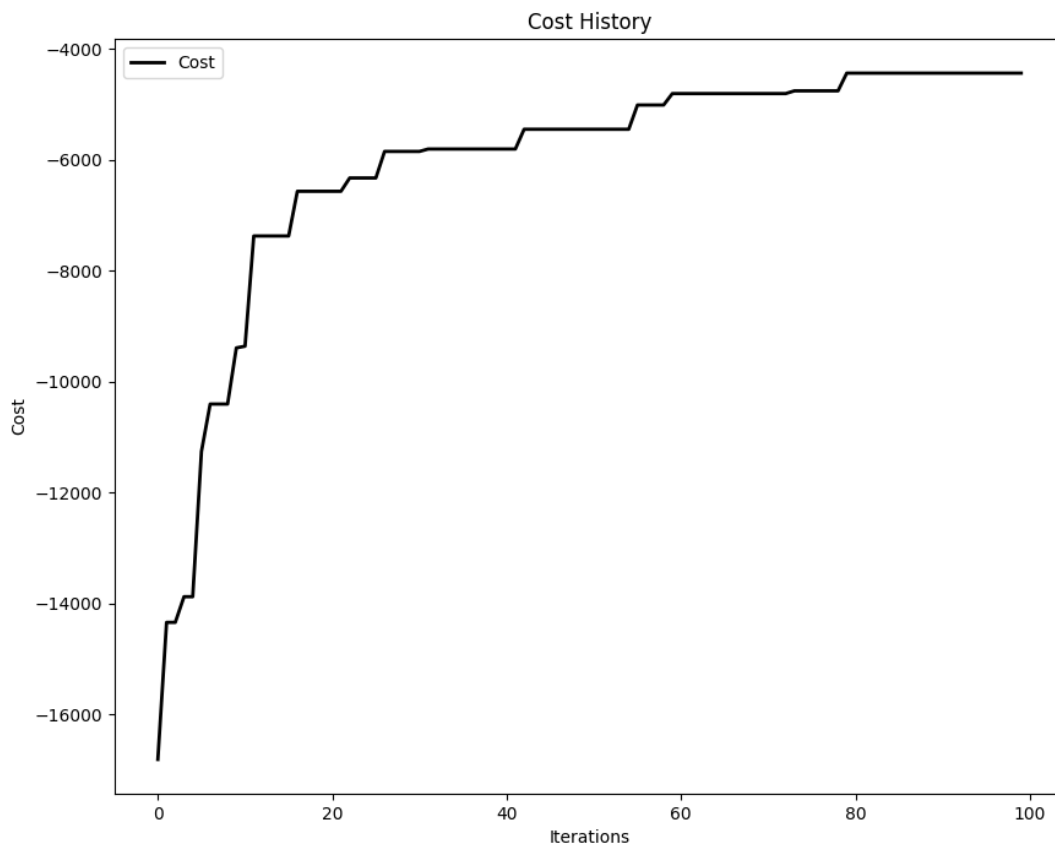
3.5 funkcja do algorytmu roju

```

def swarm_func(swarm_entities):
    n_particles = swarm_entities.shape[0]
    j = [fitness_func_swarm(swarm_entities[i]) for i in range(n_particles)]
    return numpy.array(j)

```


3.6 wyniki roju



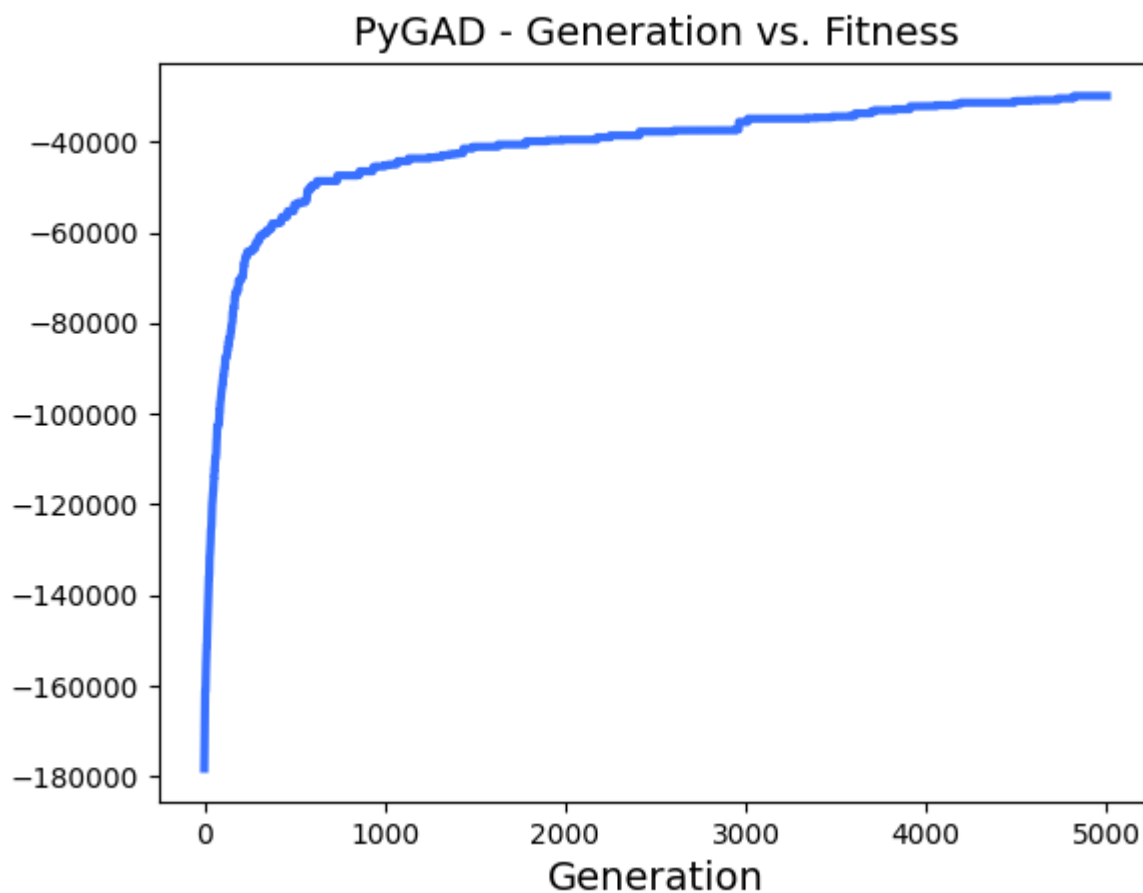
Ten algorytm generuje bardzo niski koszt, ale nie odwiedza wszystkich miast. W zależności od parametrów i metody oceniania najlepsze wyniki były zbliżone do tych z genetycznego, nie odwiedzał on często wszystkich wierzchołków, a jego wykonanie zajmowało średnio 3 razy dłużej.

4 Działanie algorytmu na dużych danych

4.1 Pierwsze testy i spostrzeżenia

Już na samym początku testów musiałem zwiększyć ilość krawędzi w grafie z $1.5 \times \text{ilość_wierzchołków}$ do $3 \times \text{ilość_wierzchołków}$, aby zbudowanie w sposób losowy grafu, w którym każde miasto jest połączone było efektywne (funkcja `make_map` zwracała wiadomość o przebudowaniu listy połączeń kilkaset razy). Potem okazało się że algorytm znajdował miasta z jednym połączeniem i krecił się w kółko uzyskując w ten sposób najlepszy dla siebie wynik według sposobu oceny. Dodałem więc założenie, że każdy wierzchołek musi mieć przynajmniej 2 sąsiadów co nie jest takie złe. W

realnej bowiem sytuacji wykorzystywalibyśmy ten algorytm do znalezienia takiej drogi pomiędzy miastami, czy punktami na mapie, więc łatwo sobie wyobrazić, że do każdego można się dostać conajmniej dwoma drogami. Ostatecznie algorytm dla 100 wierzchołków z 1000 rozwiązań na pokolenie i 5000 pokoleń wykonywał się przez 6932 sekund, czyli nieco po nad godzinę. Jednak jego wyniki nie były zachęcające.

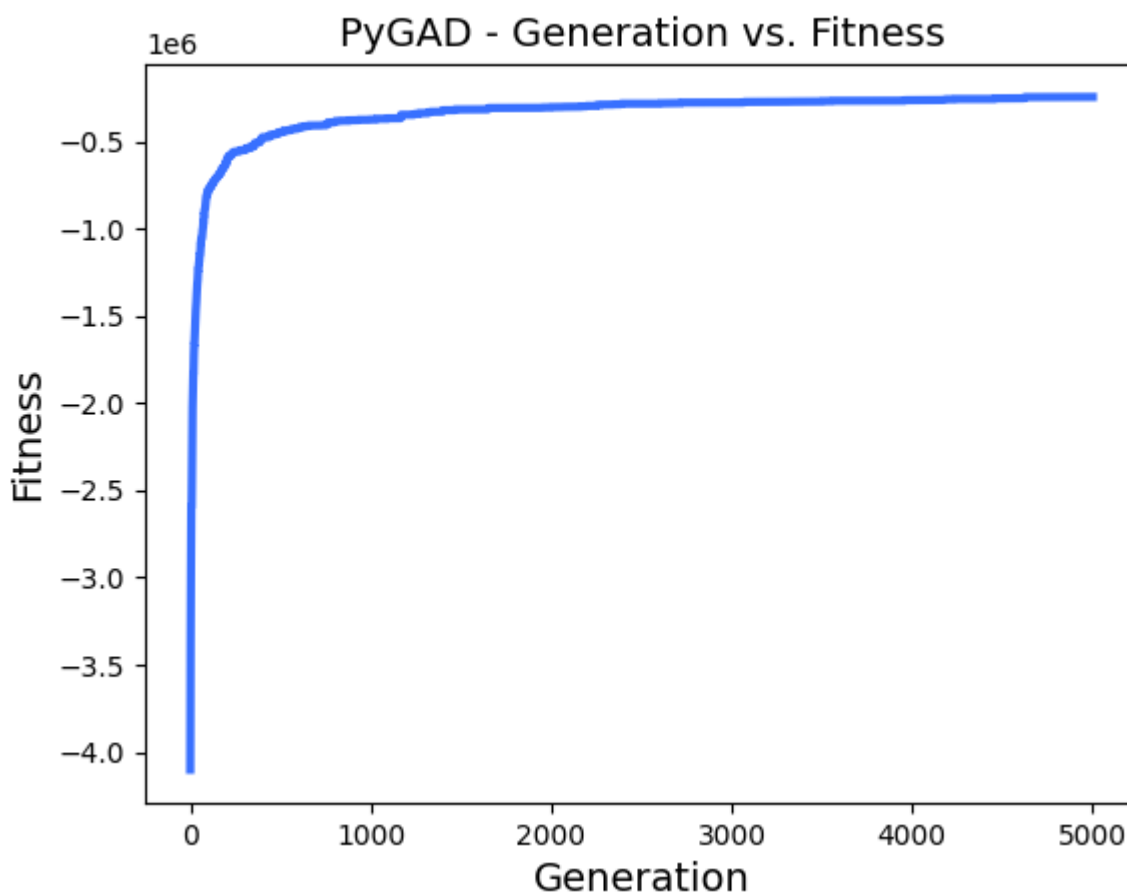


Pomimo stałego polepszenia się uzyskanej oceny uruchomienie funkcji czyszczacej wynik ujawniło, że zostały odwiedzone jedynie 53 ze 100 wierzchołków, oraz zostały wykonanych ok. 10 nielegalnych ruchów (wiadomość "poleciał samolotem"). Postanowiłem więc wprowadzić zmiany w sposobie oceniania, aby nagradzane było nie odwiedzenie wszystkich miast, ale nagroda przyznawana była za każde nowe odwiedzone miasto, oraz zwiększona została kara za nielegalne ruchy.

4.2 Zmiana w ocenianiu

```
if len(cities_visited) == number_of_cities:  
    fitness = fitness / 3  
return -int(fitness)*(number_of_cities/len(cities_visited)*5)
```

4.3 Wyniki po zmianie



Ulepszone rozwiązanie wciąż nie jest idealne, zawiera więcej nielegalnych ruchów niż poprzednie za to w zamian odwiedzonych zostało 93 ze 100 wierzchołków. W celu uzyskania idealnego wyniku prawdopodobnie należało by dokonać kolejnych zmian z funkcji fitness, zwiększyć docelową ilość ruchów z $2 \times$ ilość wierzchołków lub przeznaczyć więcej mocy obliczeniowej. To ostatnie wydaje się najmniej optymalne, bo wynik został zwrócony ponownie po trochę ponad godzinie.

5 Podsumowanie

Rozwiązywanie tego typu problemu wymaga dużo czasu i mocy obliczeniowej. Na pewno istnieją lepsze metody oceniania umożliwiające znalezienie lepszych rozwiązań w znacznie krótszym czasie niż moje. Ze względu na mój aktualny brak czasu i chęci nie próbowałem rozwiązać tego problemu z wykorzystaniem algorytmu Dijkstry (metoda brute force), ale zbierając informacje na internecie oraz od kolegów, którzy testowali podobne rozwiązania jest ono efektywne dla małych grafów, ale słabo się skaluje.

[Repo z rozwiązaniem](#)

Koniec