Znajdowanie najkrótszej drogi w grafie ważonym(Python)

Marek Wiecaszek

April 27, 2022

1 Wstep

Rozważać bedzimy problem komiwojażera, czyli bedziemy próbowali znaleźć droge przez graf ważony o najmniejszym koszcie, która odwiedza każdy wierzchołek.

Rozwiazanie takiego problemu dla małych grafów jest trywialne, można bowiem, kożystajac z klasycznych funkcji, niskim kosztem znaleźć wszystkie możliwe ścieżki po czym wybrać najlepsza. Jednak sprawa zaczyna sie komplikować dla grafów o wiekszej złożoności. Przykładowo dla grafu, ktory posiada 100 wierzchołków, a każdy z nich 3 połaczenia, liczba możliwych dróg odwiedzajacych każdy wierzchołek (zakładajac, że uda nam sie to osiagnać najmniejsza możliwa liczba ruchów) bedzie troche mniejsza od 3*2do98. Jest to liczba z 29 zerami, a liczba wykonanych operacji dla każdej drogi to co najmniej 100n.

Można jednak spróbować rozwiazać ten problem algorytamami genetycznymi, który pozwola nam znaleźć droge bliska najlepszej dużo mniejszym kosztem. W moim rozwiazaniu jest to w 2*liczba wierzchołków * liczba generacji * rozwiazania na generacje * 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 zawierajaca wierzchołki wraz z nazwa i potrzebna później funkcjonalnościa i anologicznie klase Route stanowiaca krawedzie.

```
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 funkjce odpowiedzialna za samo jego skonstruowanie

```python
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 poczatku tworzone sa losowe rozwiazania z pośród, których wybierane sa najlepsze. Nastepnie na podstawie określonej ilości najlepszych rozwiazań generowane jest kolejne pokolenie. Algorytm kończy swoje działanie po wykonaniu ustalonej ilości operacji lub osiagnieciu podanej wartości końcowej. Do oceniania wyników służy funkcja fitness (indywidualna dla każdego problemu) przedstawiona poniżej

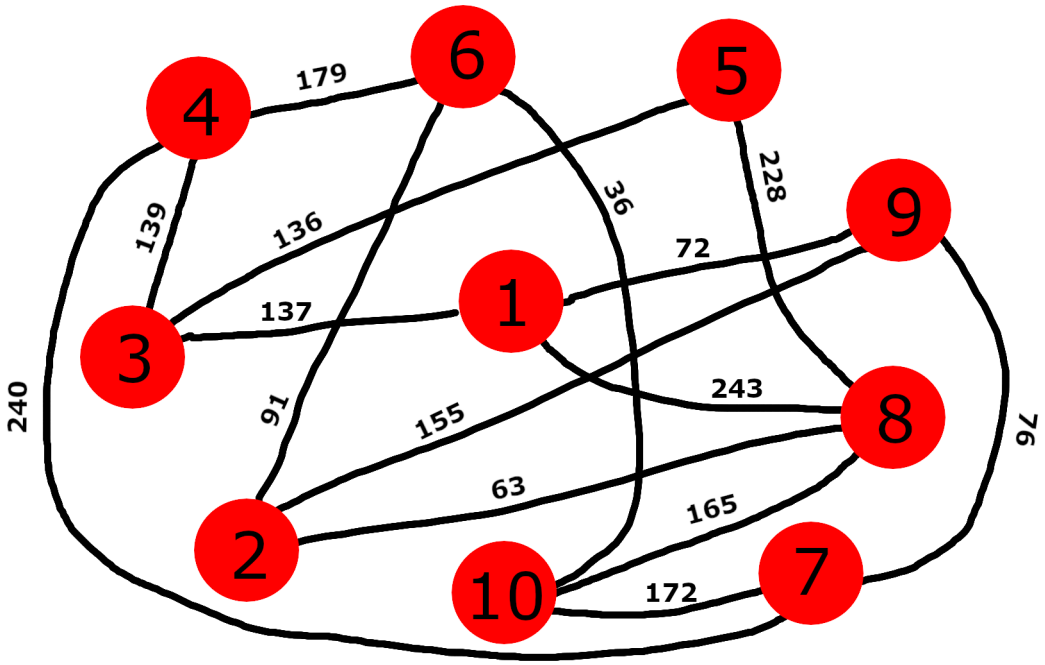
```python
def fitness_func(solution, solution_idx):
    cities_visited = [starting_point]
    previous_city = cities_1[starting_point]
    fitness = 0
    for move in solution:
        legal_moves = previous_city.connected_with_indexes(cities_1)
        if int(move) in legal_moves:
            con = 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 sie przykładowa konfiguracja wywołania algorytmu generycznego (wartości w zależności od potrzeb ulegaja 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

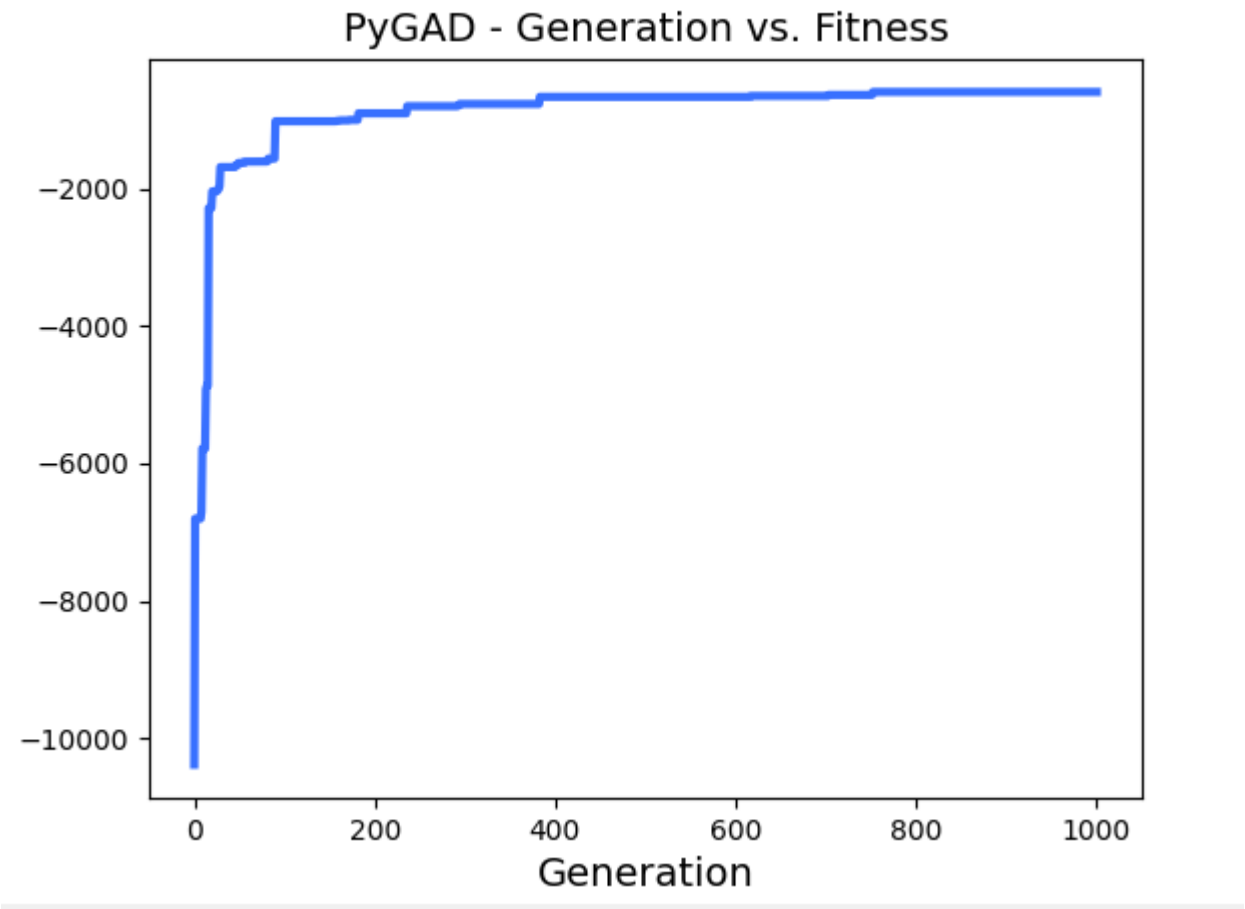
3.1 Dla przejżystości najpierw zobaczmy jak działa algorytm dla grafów, które jesteśmy w stanie objać wzorkiem



Jak widzimy powyższy graf nie jest bardzo skomplikowany i zapewne człowiek byłby w stanie znaleźć dla niego optymalne rozwiazanie w przeciagu kilku minut. Dla komputera to zadanie zajeło 7 sekund co nie wydaje sie być mocno imponujace, trzeba jednak zwrócić uwage, że aby ten problem roziwazać trzeba myśleć z czym urzadzenie nie radzi sobie z byt dobrze. Dlatego ten wynik można uznać za dość dobry.

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

3.2 Rozwój algorytmu



Wykres przedstawia wyniki zwracane przez funkjce fitness dla poszczególnych pokoleń. Jak widać różnia sie one od ostatecznego wyniku 1013. Jak został uzyskany przedstawione jest poniżej.

3.3 Czyszczenie wynikow

Oczywiście aby algorytm mógł sie rozwijać potrzebna mu była wieksza liczba ruchów niż końcowa. Przyjałem założenie, że 2*liczba wierzchołków bedzie wystarczajace. Oznacza to jednak, że na koniec bedzie trzeba te wyniki oczyścić z niepotrzebnych ruchów. Do takich zaliczamy okazjonalne niepotrzebne zapetlanie 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.

```python
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("Znalezione nielegalny ruch (poleciał samolotem)")
    print("Odwiedzone miasta: ", len(cities_visited))
    return real_weight, moves_used
```

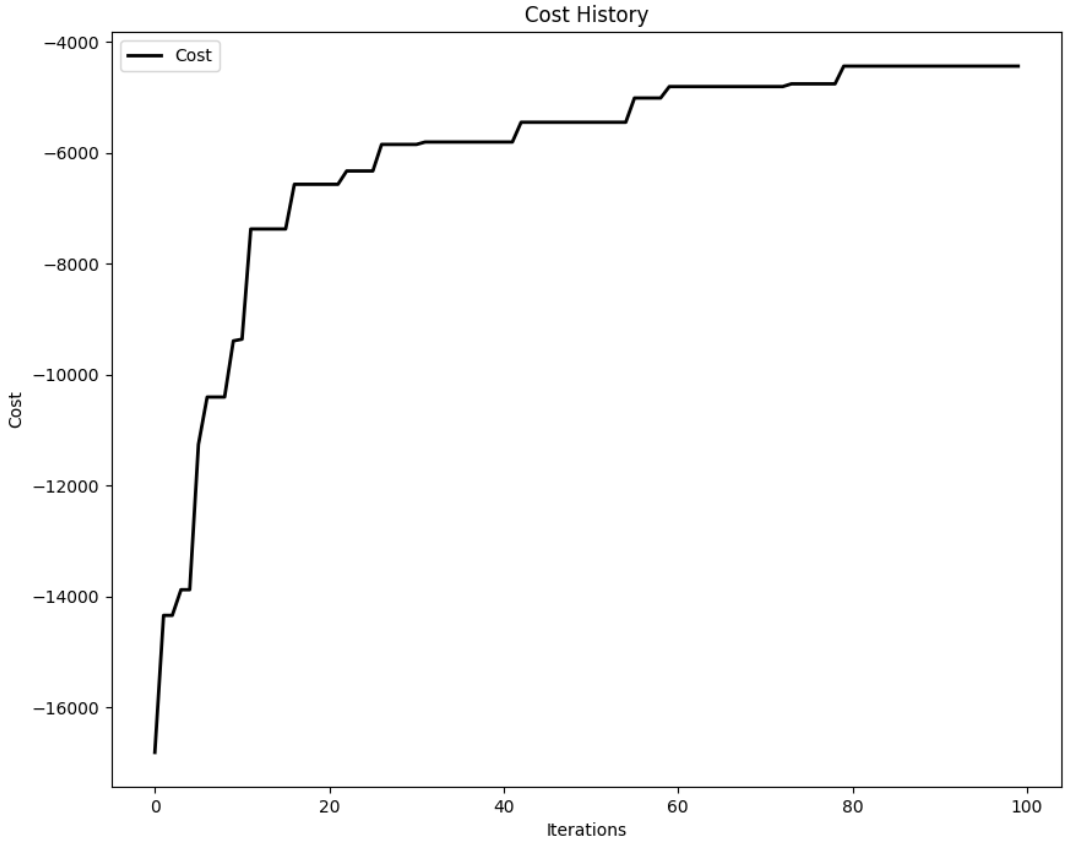
3.4 Inne rozwiazanie - metoda roju (swarm)

Innym sposobem rozwiazania tego problemu może być algorytm roju. Przedstawie go jednak bardziej jako ciekawostke, gdzyż po licznych testach okazał sie być gorszym wyborem.

3.5 funkcja do algorytmu roju

```python
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



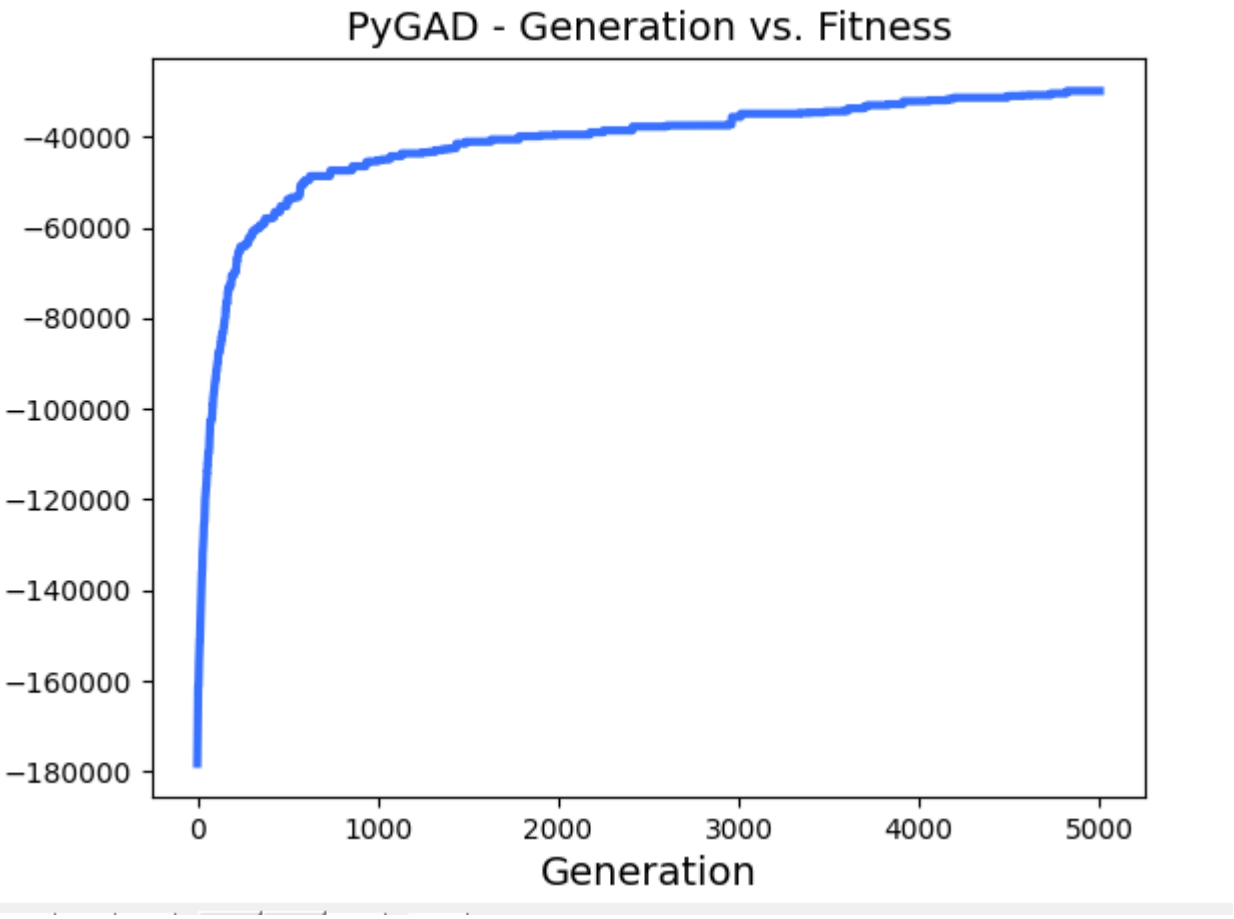
Ten algorytm generuje bardzo niski koszt, ale nie odwiedza wszystkich miast. W zależności od parametrów i metody oceniania najlepsze wyniki były zbliżone do tych z genetycznego, nie odwiedzał on czesto wszystkich wieżchołków, a jego wykonanie zajmowało średnio 3 razy dłużej.

4 Działanie algorytmu na dużych danych

4.1 Pierwsze testy i spostrzeżenia

Już na samym poczatku testów musiałem zwiekszyć ilość krawedzi w grafie z 1.5*ilość_wieszchołków do 3*ilość_wieszchołków, aby zbudowanie w sposób losowy grafu, w którym każde miasto jest połaczone było efektywne(funkcja make_map zwracała wiadomość o przebudowaniu listy połaczeń kilkaset razy). Potem okazało sie że algorytm znajdywał miasta z jednym połaczeniem i krecił sie w kółko uzyskujac w ten sposób najlepszy dla siebie wynik według sposobu oceny. Dodałem wiec założenie, że każdy wierzchołek musi mieć przynajmniej 2 sasiadów co nie jest takie złe. W

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

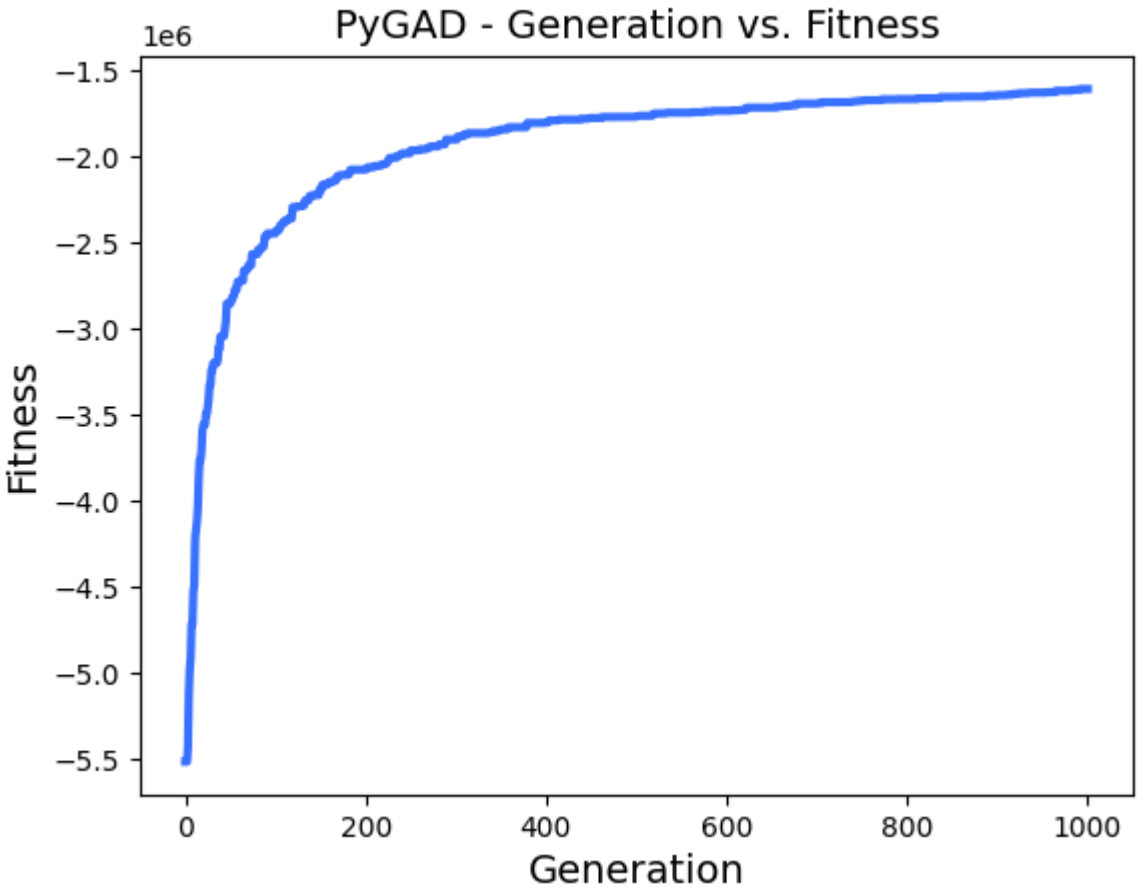


Pomimo stałego polepszenia sie uzyskanej oceny uruchomienie funkcji czyszczacej wynik ujawniło, że zostały odwiedzone jedynie 53 ze 100 wierzchołków, oraz zostały wykonanych ok. 10 nielegalnych ruchów (wiadomość "poleciał samolotem"). Postanowiłem wiec wprowadzić zmiany w sposobie oceniania, aby nagradzane było nie odwiedzenie wszystkich miast, ale nagroda przyznawana była za każde nowe odwiedzone miasto, oraz zwiekszona 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 rozwiazanie wciaż nie jest idealne, zawiera wiecej nielegalnych ruchów niż poprzednie za to w zamian odwiedzonych zostało 97 ze 100 wierzchołków. W celu uzyskania idealnego wyniku prawdopodobnie należało by dokonać kolejnych zmian z funkcji fitness lub przeznaczyć wiecej mocy obliczeniowej. Aby algorytm działał w pełni poprawnie najważniejsze jest znalezienie złotego środka w sposobie oceny. Do tego wykonania dodatkowo zwiekszona została liczba dozwolonych ruchów kosztem mniejszej ilości pokoleń co skóricło czas do "zaledwie" 38 minut. wnioskujac jednak po wykresie wynik miał tendencje do dalszego polepszania sie, wiec ponowne zwiekszenie liczby generacji pozowiłoby uzyskać korzystniejszy wynik

5 Podsumowanie

Rozwiazywanie tego typu problemu wymaga dużo czasu i mocy obliczeniowej. Na pewno istnieja lepsze metody oceniania umożliwiajace znalezienie lepszych rozwiazań w znacznie krótszym czasie niż moje. Ze wzgledu na mój aktualny brak czasu i checi nie próbowałem rozwiazać tego problemu z wykorzystaniem algorytmu Dijkstry (metoda brute force), ale zbierajac informacje na internecie oraz od kolegów, którzy testowali podobne rozwiazania jest ono efektywne dla małych grafów, ale słabo sie skaluje.

Repo z rozwiazaniem

Koniec