Sprawozdanie z projektu

Wiktor Płużek

Spis treści

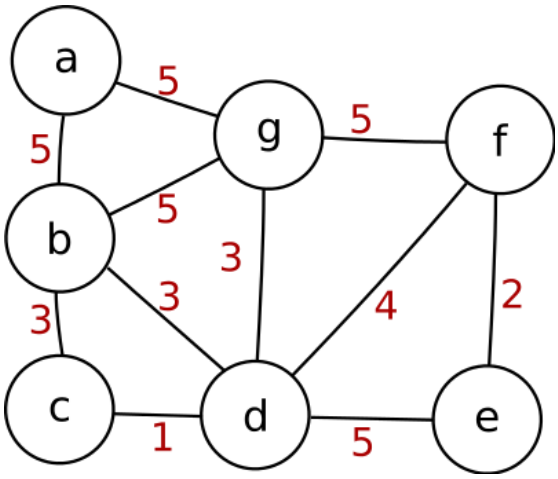
Cel projektu

Celem projektu było zapoznanie się z różnymi algorytmami genetycznymi oraz zastosowanie ich w celu rozwiązania zadanych problemów. Wybranym przeze mnie problemem był problem Komiwojażera (travelling salesman).

Opis problemu

Problem komiwojażera polega na znalezieniu ścieżki z najmniejszą sumą wag, która przejdzie przez wszystkie wierzchołki grafu.

Dla przykładowego grafu G:



Dla podanego grafu, algorytm powinien znaleźć najkrótszą ścieżkę przejścia przez wszystkie wierzchołki. W tym przypadku jest to a -> b -> c -> d -> g -> f -> e

Generowanie grafu

Do celów generowania losowego grafu, napisałem funkcję make_random_graph opierającą się na paczce networkx.

```
def make_random_graph(n, m):

    def generate_graph(n, m):
        array = []
        for i in range(n):
            array.append(i)
        graph = {}
        for i in array:
            graph[i] = sorted(random.sample(range(0, n), m))
        return graph

    g = nx.Graph(generate_graph(n, m))
    for (u, v) in g.edges():
        g.edges[u, v]['weight'] = random.randint(0, 10)

    g.remove_edges_from(nx.selfloop_edges(g))

    return g
```

Za pomocą tej funkcji generowałem grafy, które później poddawałem optymalizacji przez algorytmy genetyczne.

Implementacja algorytmem genetycznym

Geny i długość chromosomu

Do obsługi algorytmów genetycznych skorzystałem z pomocy paczki pygad. Algorytm genetyczny składa się z genów, które zawierały numery wierzchołków. Pierwszy problem pojawia się przy doborze długości chromosomu – mogą one zawierać dokładnie tyle genów ile graf ma wierzchołków, co spowodowałoby, że jedynym możliwym do wygenerowania przez nie rozwiązaniem byłaby najkrótsza ścieżka która przechodzi przez każdy wierzchołek dokładnie raz. Jednak, czasami połączenia między wierzchołkami tak się układały, że tańszą

ścieżką byłoby wrócenie się i pójście inną drogą. Biorąc to pod uwagę, postanowiłem ustawić długość dwukrotnie dłuższą niż ilość wierzchołków, aby dać algorytmowi nieco możliwości ruchu.

```
gene_space = {'low': 0, 'high': i, 'step': 1}
num_genes = len(graph.nodes)*2
```

Funkcja fitness

Następną ważną rzeczą przy optymalizacji algorytmami genetycznymi jest funkcja oceniająca otrzymane próbki genów. Najprostszym sposobem było napisać funkcję, która sprawdzała czy wygenerowane przez nie ścieżki są prawidłowe. Początkowe zasady funkcji fitness:

- Karanie jeśli ścieżka jest nieprawidłowa – brak przejścia między danymi węzłami
- Liczenie wagi przejścia – niższa waga całkowitej drogi to lepszy wynik

Implementacja:

```
def fitness_func(solution, solution_idx):
    fitness = 0
    position = int(solution[0])
    for i in solution:
        i = int(i)
        if i in nx.to_dict_of_lists(graph)[position]:
            fitness -= nx.get_edge_attributes(graph,
'weight')[min(position, i), max(position, i)]
            position = i
        else:
            fitness -= len(graph.nodes) * 2
    return fitness
```

Po zastosowaniu takiej funkcji, zaczęły się pojawiać pierwsze rozwiązania, jednak były one dalekie od poprawnych. Większość rozwiązań zamykała się w pętli tanich przejść, aby zminimalizować zebrane wagi, co skutkowało brakiem przejścia przez wszystkie węzły. Trzeba było więc wymusić przejście przez wszystkie węzły.

Aby to zrobić, stworzyłem tabelę, która przechowywała odwiedzone już węzły, a jeśli jej końcowa długość nie odpowiadała ilości węzłów w grafie, surowo karałem dane rozwiązanie.

Dodatkowo, aby zapewnić faktyczną wagę wszystkich przejść, w momencie gdy lista visited[] była równa liście wszystkich wierzchołków, funkcja przestawała obliczać dalsze przejścia, bo jedynie pogorszyłoby to wyniki już kompletnego rozwiązania.

```
def fitness_func(solution, solution_idx):
    fitness = 0
    visited = [int(solution[0])]
    position = int(solution[0])
    for i in solution:
        if len(visited) is len(graph.nodes):
            return fitness
        i = int(i)
        if i in nx.to_dict_of_lists(graph)[position]:
            if i in visited:
                fitness -= nx.get_edge_attributes(graph,
'weight')[min(position, i), max(position, i)]
                position = i
            else:
```

```
                visited.append(i)
                fitness -= nx.get_edge_attributes(graph,
'weight')[min(position, i), max(position, i)]
                position = i
        else:
            fitness -= len(graph.nodes) * 2
    if len(visited) is not len(graph.nodes):
        fitness -= len(graph.nodes) * 10
    return fitness
```

Dzięki temu program zaczął zwracać ścieżki, które przechodziły przez wszystkie węzły w grafie.

W celu dalszej optymalizacji można by karać program za kilkukrotne odwiedzanie tego samego wierzchołka, jednak uznałem, że to może odrzucać rozwiązania, które korzystały z tego ze względu na tańsze przejścia w niektórych miejscach.

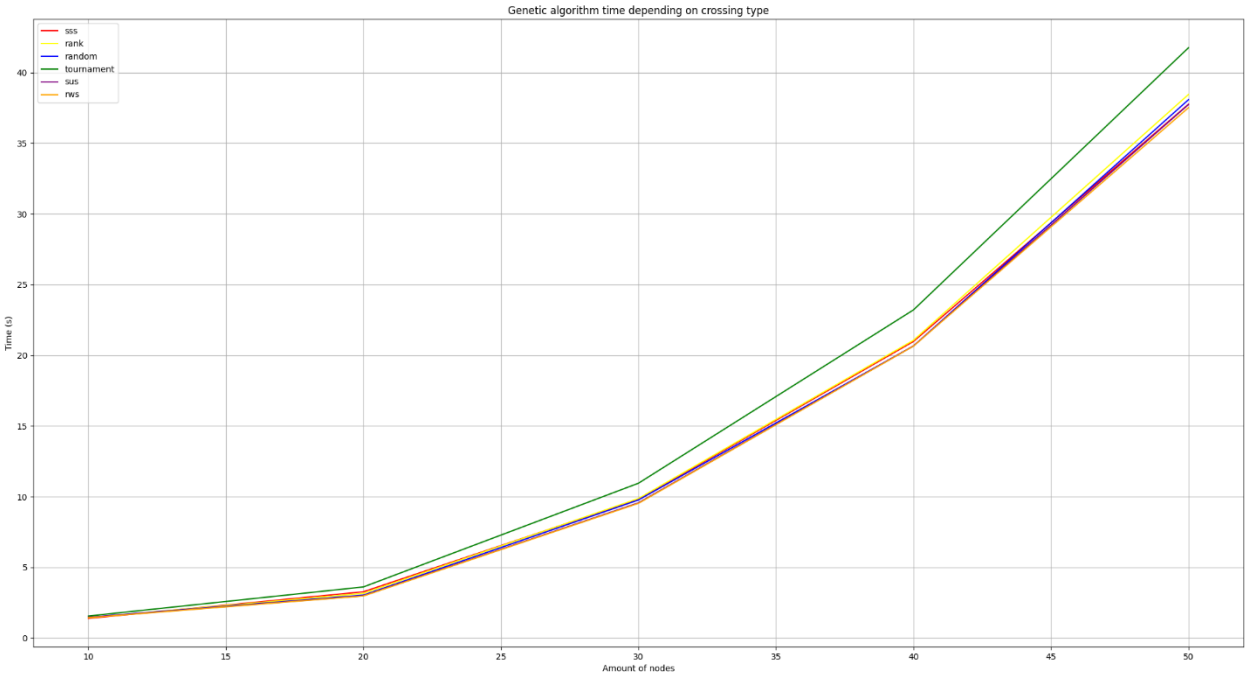
Z powodu, że algorytm genetyczny z biblioteki pygad wyszukuje maksymalnego rozwiązania, musiałem odejmować wartości wag, zamiast je dodawać, aby największy wynik był najlepszy.

Porównanie różnych sposobów wyboru rodziców

W celu dalszej optymalizacji, porównałem wiele sposobów doboru rodziców dla rozwiązań. Przy ustawieniu:

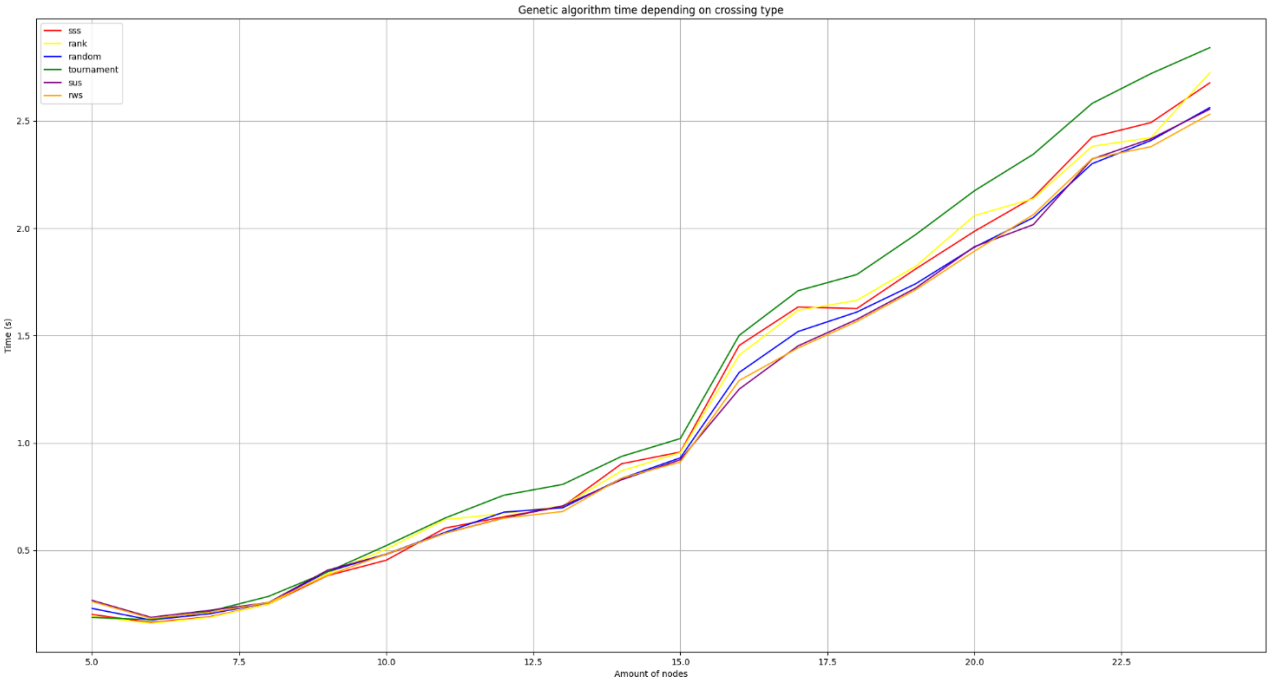
```
sol_per_pop = 50
num_parents_mating = 5
num_generations = 100
keep_parents = 3
```

Wykonałem po 5 pomiarów dla ilości węzłów w grafie od 10 do 50, oraz sprezentowałem wyniki na wykresie:



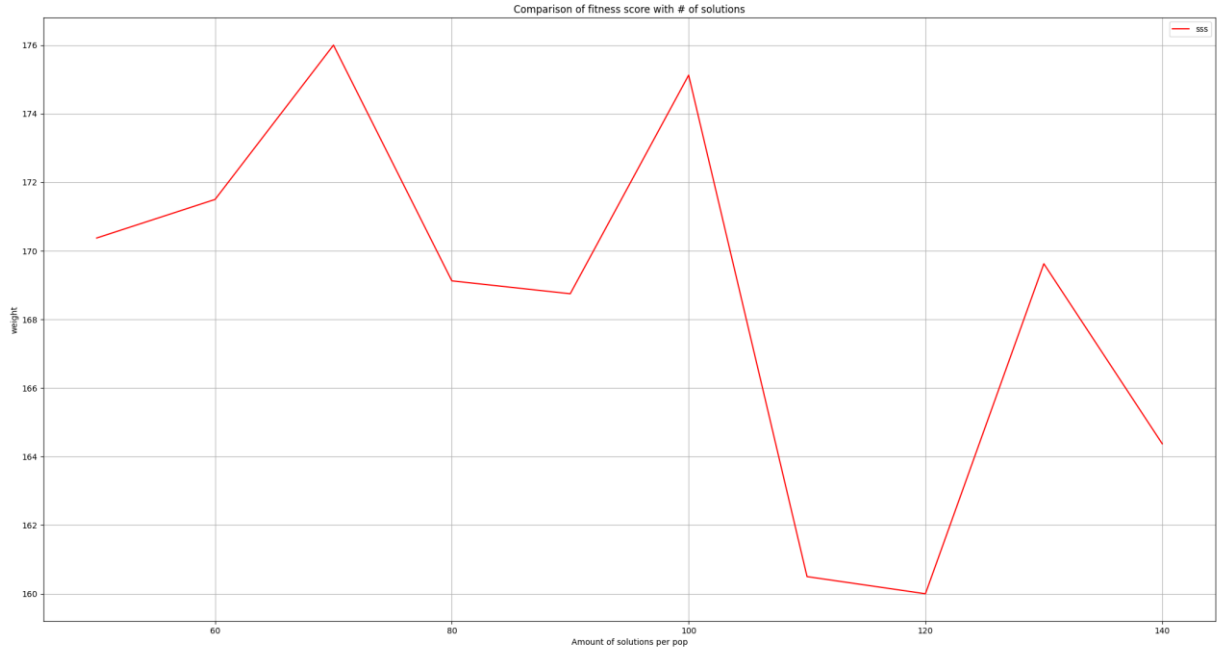
Z wykresu można zauważyć, że wszystkie sposoby wyboru rodziców były w miarę podobne czasowo, poza sposobem tournament, który znacząco odstawał od reszty – w związku z tym w dalszym użytkowaniu algorytmu genetycznego postanowiłem pozostać przy doborze metodą steady_state_selection (sss), ponieważ działała ona na równi z pozostałymi, a nie wprowadzała losowości jak niektóre z nich.

Kolejny wykres dla mniejszej wielkości grafu, ale z większą ilością prób na każdą wielkość grafu pokazuje podobną zależność – sposób tournament odstaje od pozostałych

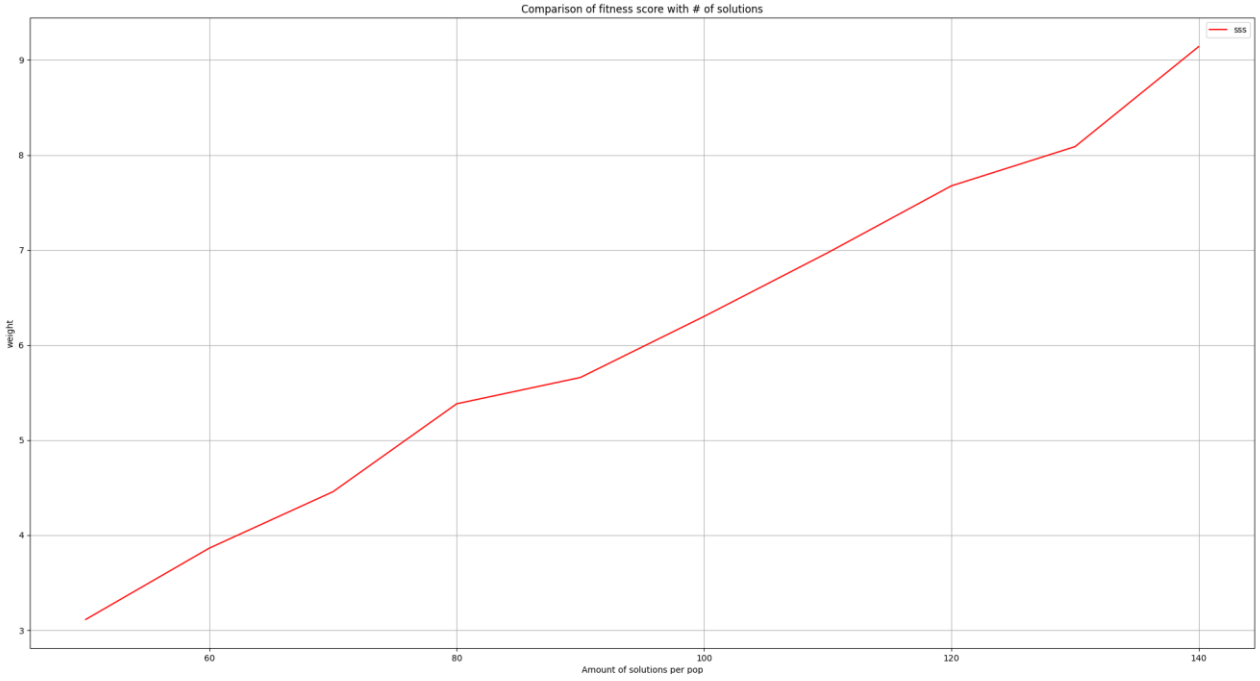


Czas działania a ilość populacji i generacji

Można się spodziewać, że ilość populacji jak i generacji powinna pozytywnie wpłynąć na wynik najlepszego rozwiązania, jednak z pewnością wpłyną one negatywnie na czas działania programu



Na wykresie widać, ze pomimo lekkiej losowości, wraz z większą ilością solutions per pop, spada średnia waga optymalnego rozwiązania



Niestety cierpi na tym czas wykonania programu, w stosunku dużo bardziej liniowym niż spadek wag rozwiązania. A więc zgadza się to z przewidywaniami.

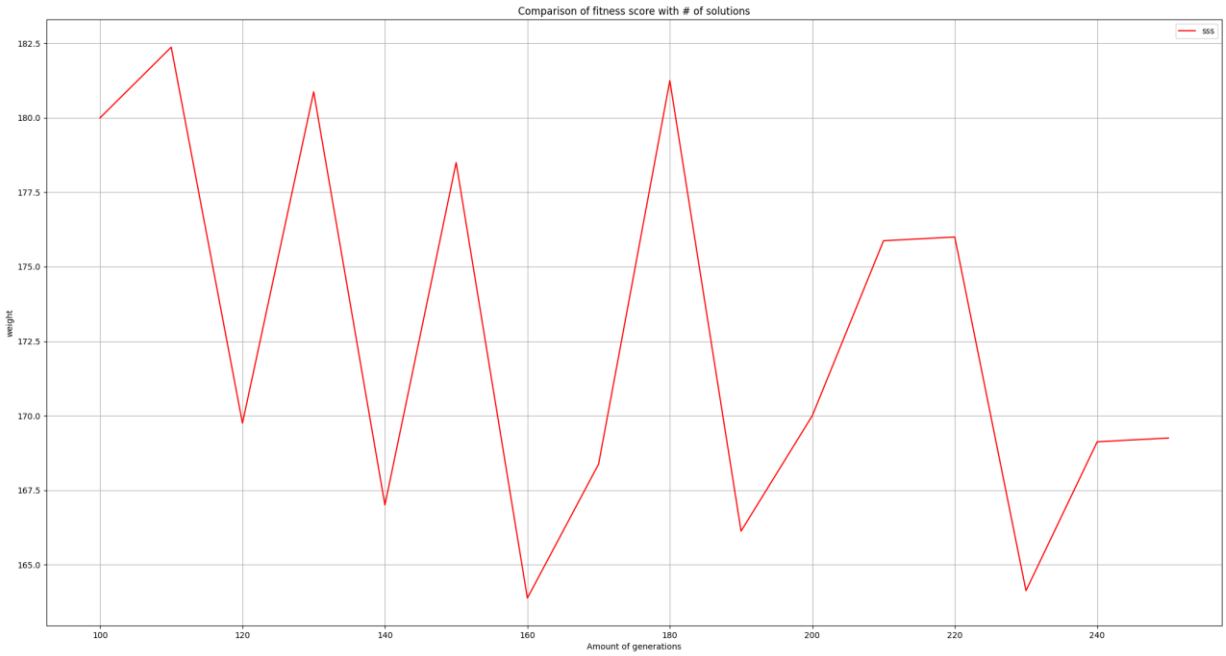
Dla tych testów były zachowane następujące ustawienia:

```
sol_per_pop = i
num_parents_mating = sol_per_pop//5
num_generations = 100
keep_parents = num_parents_mating//3
parent_selection_type = "sss"
crossover_type = "single_point"
```

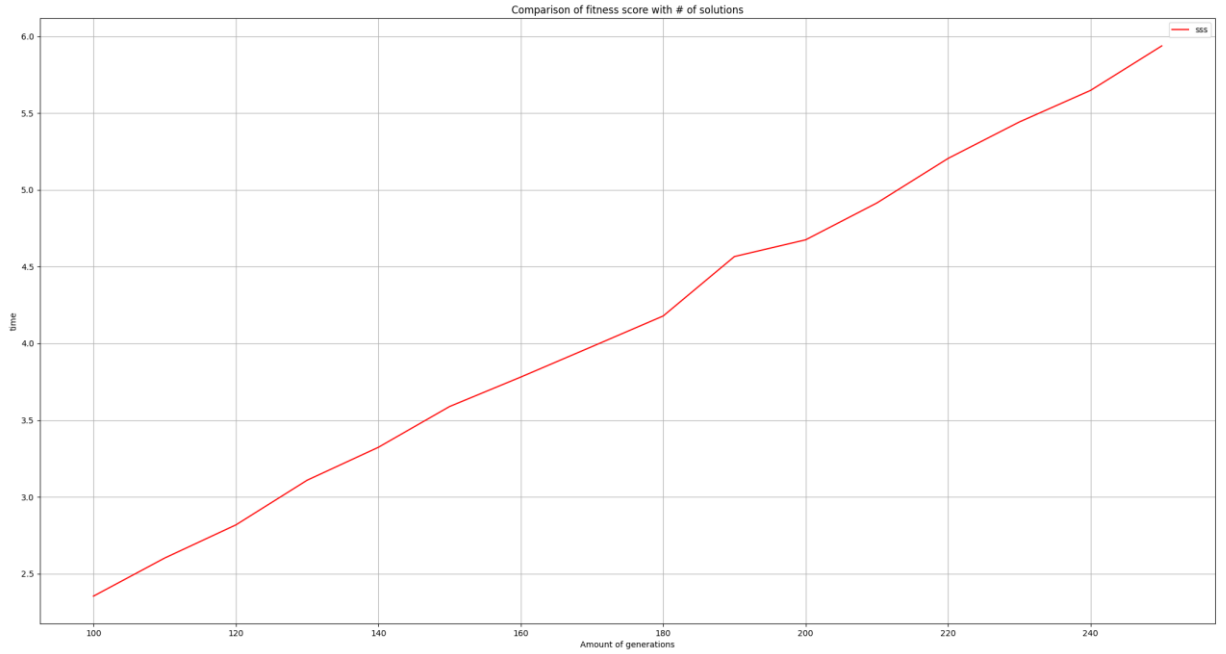
```
mutation_type = "random"
mutation_percent_genes = 10
gene_space = {'low': 0, 'high': 11, 'step': 1}
```

Gdzie sol_per_pop było w zakresie od 50 do 140 ze skokiem co 10.

Spójrzmy teraz na działanie programu dla różnych ilości generacji



Jak widać na wykresie, wraz z większą ilością generacji, utrzymuje się tendencja spadkowa optymalnego wyniku, niestety tak jak w przypadku ilości rozwiązań na generację, ma to negatywny wpływ na czas działania programu



Tutaj ustawienia były podobne jak dla wcześniejszych testów.

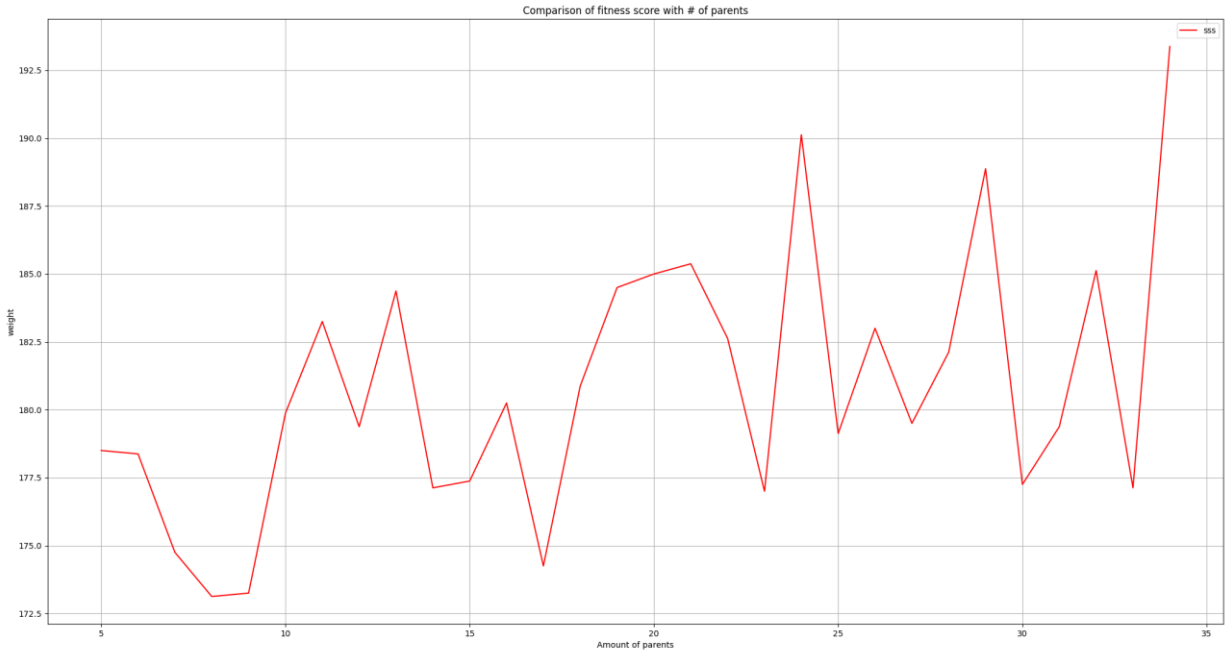
```
sol_per_pop = 50
num_parents_mating = sol_per_pop//5
```

```
num_generations = i
keep_parents = num_parents_mating//3
parent_selection_type = "sss"
crossover_type = "single_point"
mutation_type = "random"
mutation_percent_genes = 10
gene_space = {'low': 0, 'high': 11, 'step': 1}
```

Gdzie num_generations było w zakresie od 100 do 250 ze skokiem co 10

Ilość rodziców a wynik

Można zadać pytanie jak ilość rodziców wpływa na wynik dla danej populacji



Na powyższym wykresie widać, że jest to w zasadzie losowe, dla tych danych została ustalona populacja 50, ilość generacji równa 100, a ilość rodziców, którzy zostawali, równa 1/3 ilości wszystkich rodziców. Testy były przeprowadzane ośmiokrotnie na grafie o 12 wierzchołkach.

Wniosek

Algorytmy genetyczne są w stanie rozwiązać problem komiwojażera, podając ścieżkę równą lub bliską najoptymalniejszej ścieżce. Problemem staje się czas działania programu, gdyż wraz z większą ilością populacji czy generacji, co zwiększa szansę otrzymania lepszego wyniku, znacznie wzrasta czas wykonania. Słabo radzą sobie one również z większą ilością wierzchołków grafów – zwiększa to długość chromosomu, jak i ilość możliwości genów.

Implementacja algorytmem roju cząstek

Przestrzeń wartości cząstek

Do obsługi roju cząstek skorzystam z pomocy paczki pyswarms, oraz wbudowanej w nią funkcję GlobalBestPSO

Jako przestrzeń wartości cząstek w roju cząstek, również ustawiam możliwe wierzchołki grafu, oraz daję cząstkom możliwość wybrania ścieżki dwukrotnie dłuższej niż ilość węzłów w grafie.

```
x_max = np.ones(len(graph.nodes) * 2) * len(graph.nodes)
x_min = np.zeros(len(graph.nodes) * 2)
my_bounds = (x_min, x_max)
```

Parametry globalnego i lokalnego oddziaływania cząstek oraz zmianę prędkości cząstek do testów ustawiłem następująco

```
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
```

Przy wywołaniu funkcji optimizer.GlobalBestPSO, wyznaczam ilość wymiarów tak jak i przestrzeń wartości, na dwukrotnie większą od węzłów grafu

```
optimizer = ps.single.GlobalBestPSO(n_particles=50,
dimensions=len(graph.nodes) * 2, options=options, bounds=my_bounds)
```

Funkcja fitness

Funkcja fitness, jest identyczna jak w algorytmie genetycznym, jedyna różnica zachodzi w tym, że zamiast odejmować wagi, to je dodajemy, ponieważ funkcja GlobalBestPSO szuka minimum, a nie maksimum. Dodatkowo, należy funkcję fitness zagnieździć w funkcji f, która uruchomi funkcję fitness na każdej z cząstek roju.

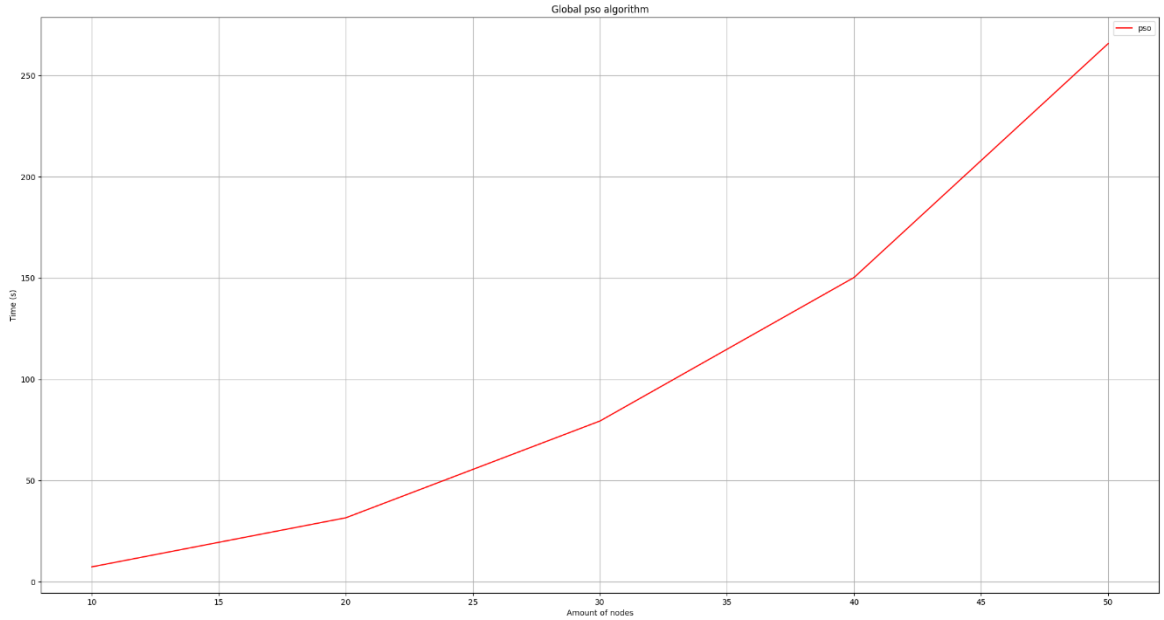
```
def fitness_func(solution):
    fitness = 0
    visited = [int(solution[0])]
    position = int(solution[0])
    for i in solution:
        i = int(i)
        if len(visited) is len(graph.nodes):
            return fitness
        if int(i) in nx.to_dict_of_lists(graph)[position]:
            if i in visited:
                fitness += nx.get_edge_attributes(graph,
'weight')[min(position, i), max(position, i)]
                position = i
            else:
                visited.append(i)
                fitness += nx.get_edge_attributes(graph,
'weight')[min(position, i), max(position, i)]
                position = i
        else:
            fitness += len(graph.nodes) * 2
    if len(visited) is not len(graph.nodes):
        fitness += len(graph.nodes) * 10
    return fitness


def f(x):
```

```
    n_particles = x.shape[0]
    j = [fitness_func(x[i]) for i in range(n_particles)]
    return np.array(j)
```

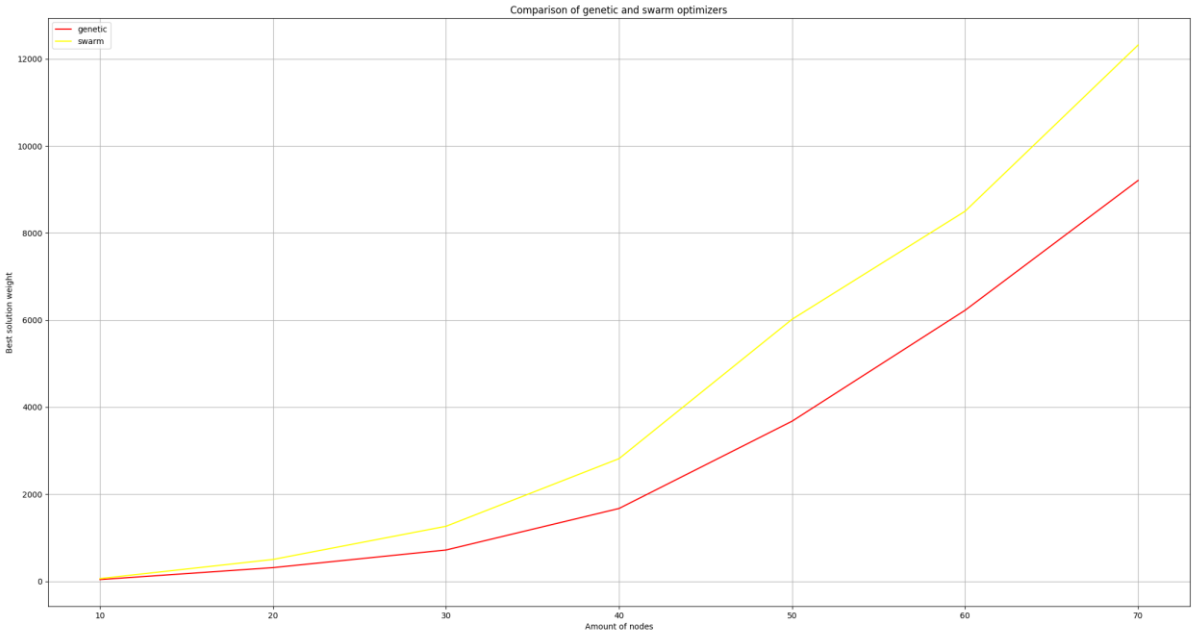
Cząstki roju a ilość węzłów

Podobnie jak w przypadku algorytmu genetycznego, czas wykonania rośnie wraz z ilością węzłów w grafie



Rośnie on jednak znacznie szybciej niż dla algorytmu genetycznego, co może być spowodowane coraz większą ilością przestrzeni co staje się ciężkie do obliczenia.

Co więcej, okazuje się, że produkuje on również gorsze wyniki niż algorytm genetyczny



Może to być spowodowane relatywnie niską ilością cząstek i iteracji – 100 cząstek w 200 przebiegach, jednak nawet tak niskie wartości są wolniejsze niż algorytm genetyczny

Wniosek

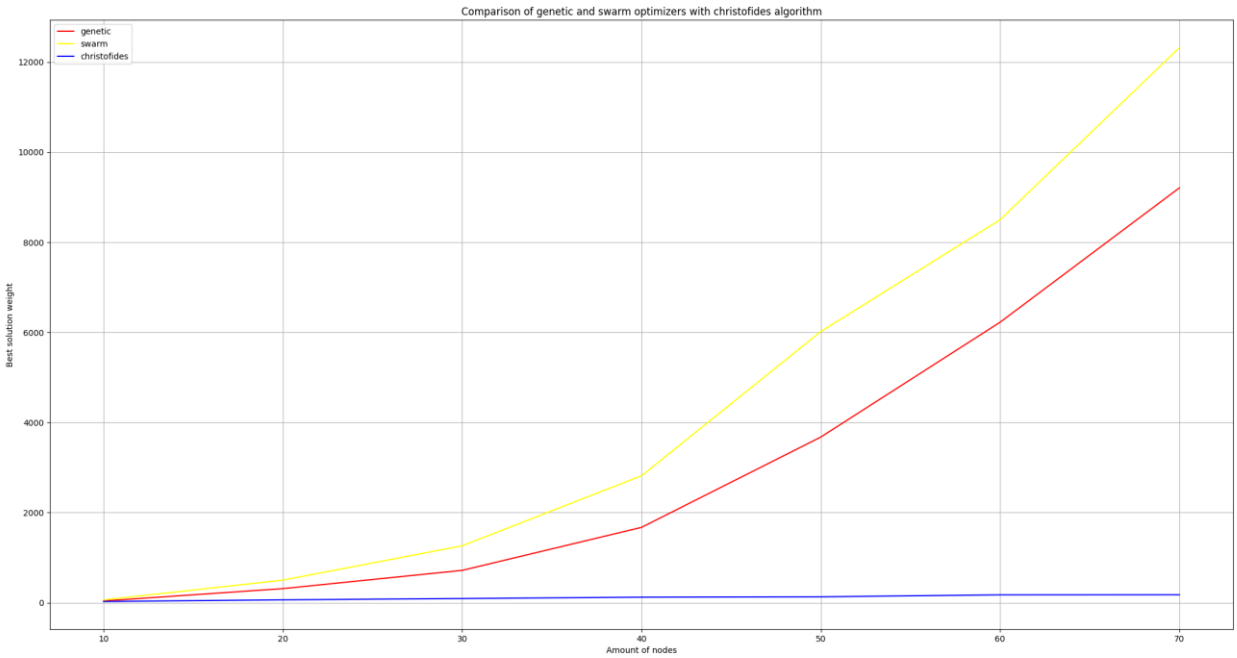
Algorytm cząstek roju również może rozwiązać problem komiwojażera, jednak bardzo dla bardzo niewielkich ilości węzłów – przy większych ilościach staje się niewydajny czasowo oraz daje dalekie od najlepszego rozwiązania.

Implementacja algorytmem heurystycznym

Algorytm christofides z biblioteki networkx

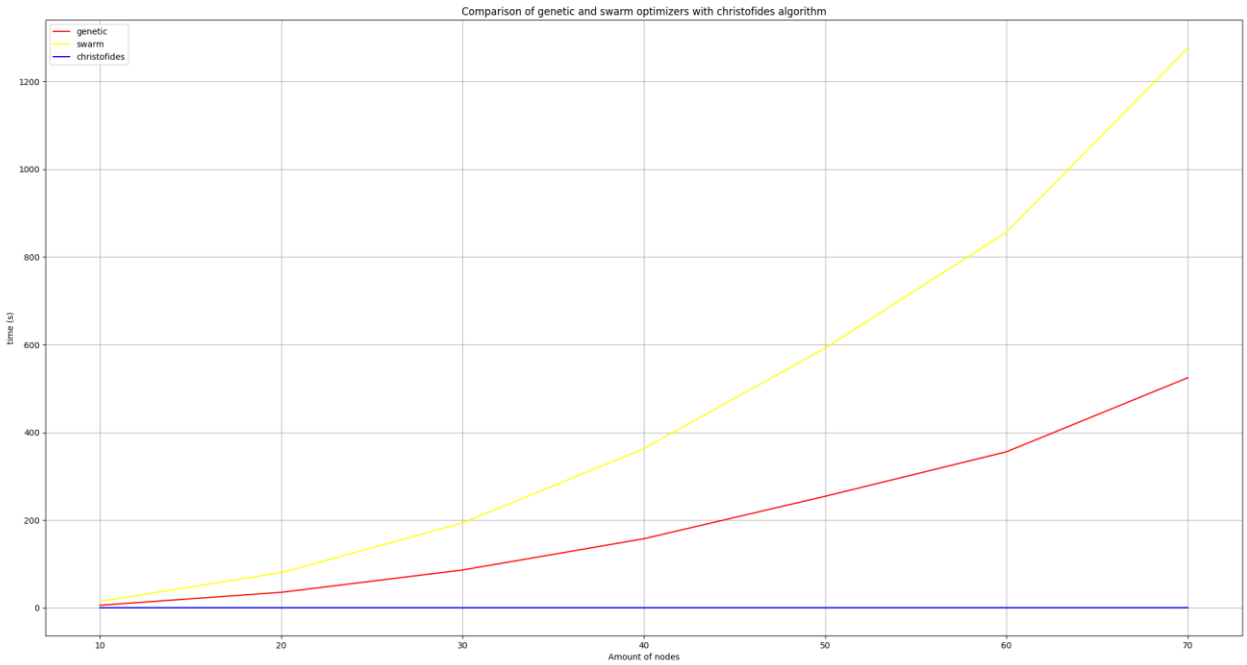
Na koniec, chciałbym porównać wyniki algorytmu genetycznego i roju cząstek z wynikiem algorytmu christofidesa wbudowanego do biblioteki networkx. Jest to algorytm o podejściu heurystycznym, o optymalności 1.5x, zatem wyniki algorytmu nie będą gorsze od rzeczywistej najkrótszej ścieżki więcej niż 1.5x.

Porównanie z algorytmami genetycznymi



Na wykresie można zobaczyć, że zarówno algorytm cząstek roju jak i genetyczny znacznie odbiegają od aproksymacji algorytmem christofidesa. Dla 60 węzłów w grafie, wynik cząstek roju wynosił 8500, wynik algorytmu genetycznego 6225, a wynik algorytmu christofidesa zaledwie 175.

Dodatkowo, oba algorytmy miały o wiele dłuższy czas działania, dla 60 węzłów, czas cząstek roju wynosił 856 sekund – ponad 14 minut, algorytmu genetycznego 355 sekund – niecałe 6 minut, a algorytmu christofidesa 0.08 sekundy.



Wnioski końcowe

Algorytmy genetyczne są ciekawym sposobem na rozwiązanie wielu problemów. Nie wymagają one zagłębienia w skomplikowane matematyczne sposoby obliczania tych problemów, choć wiedza na ich temat z pewnością może pomóc. Niestety mają one również swoją wadę, którą jest niska wydajność czasowa względem algorytmów heurystycznych, oraz gorsze od nich wyniki. Końcowo oba algorytmy były w stanie wyznaczyć krótką trasę przechodzącą po wszystkich wierzchołkach dla niskich ilości węzłów. Jednak nie dla każdego problemu istnieje proste rozwiązanie matematyczne, są takie które mają jedynie rozwiązania NP trudne, wtedy można rozważyć zastosowanie algorytmów genetycznych w celu bardziej optymalnego znalezienia rozwiązania.