Problem STSP z uwzględnieniem kolejności przechodzenia wierzchołków

Data	Status projektu	Uwagi
2020–12–14	Wybór tematu	
2021–01–08	Rozpoczęty	
2021–02–12	Testowany	
2021–02–13	Gotowy do recenzji	
2021–02–14	Recenzowany i poprawiony	
2021–02–15	Wersja na DokuWiki	
2021–02–15	Gotowy do oceny	

1 Autorzy

Mateusz Kaźmierski 143942
Jakub Łepek 146470

2 Recenzent

Bartosz Chazan

2.1 Uwagi

Wyczerpujący — działa.
Losowy — działa.
Genetyczny — działa.
Można było porównać wyniki genetycznego do zachłannego (poprawione).
Testy wykonane u mnie pokazują takie same trendy jak przedstawione w tej pracy.
Ciekawa metodologia w rozdziale 11.2, jednak tak samo jak autorzy, nie jestem w stanie sprawdzić jej poprawności.
Nie mam więcej zastrzeżeń.

3 Streszczenie

Projekt opisuje metaheurystyczne rozwiązanie symetrycznego problemu komiwojażera, w którym każda waga krawędzi zostaje przemnożona przez odpowiedni skalar, w zależności od kolejności odwiedzenia. Skorzystano z algorytmu genetycznego z procedurą poprawiania rozwiązania i wykonano testy różnych cech tego algorytmu.

3.1 Słowa kluczowe

- Problem obliczeniowy
- Klasa złożoności obliczeniowej NP
- Graf
- Graf pełny
- Przestrzeń Euklidesowa
- Układ współrzędnych kartezjańskich
- Heurestyka
- Algorytm aproksymacyjny
- Metaheurestyka
- Cykl Hamiltona
- Problem komiwojażera
- Krzyżowanie
- Mutacja
- Naturalna selekcja
- Ewolucja
- Algorytm ewolucyjny
- Algorytm genetyczny
- Odchylenie standardowe
- Nierówność Czybyszewa

4 Wykorzystane technologie

- Python
- Matplotlib
- NumPy
- ImageIO

4.1 Zestawienie licencji

- Python
- Matplotlib
- NumPy
- ImageIO
- Nasz kod

5 Wstęp

Nie wszystkie **problemy obliczeniowe** można rozwiązać w **satysfakcjonującym czasie**, dlatego często stosowane są algorytmy, które produkują jedynie **przybliżone** rozwiązanie problemu albo **nie dają stuprocentowej gwarancji sukcesu**. Jeżeli taki algorytm oferuje **gwarancję jakości**, to nazywamy go **algorytmem aproksymacyjnym** [1]. Do określenia jakości rozwiązania dla danej instancji problemu, możemy zdefiniować **funkcję celu**[1]. Gwarancja jakości oznacza spełnienie jednego z warunków 1, 2, 3 [2, 3].

$$\exists_{\epsilon\in\mathbb{R}} \left|A\left(I\right) - OPT\left(I\right)\right| < \epsilon \tag{1}$$

$$\exists_{\epsilon\in(0,\infty)} \frac{A\left(I\right)}{OPT\left(I\right)} \leqslant 1+\epsilon,\ \text{gdy } A\left(I\right) \geqslant OPT\left(I\right) \tag{2}$$

$$\exists_{\epsilon\in(0,\infty)} \frac{A\left(I\right)}{OPT\left(I\right)} \leqslant 1-\epsilon,\ \text{gdy } A\left(I\right) \leqslant OPT\left(I\right) \tag{3}$$

I — instancja problemu; A(x) — rozwiązanie dla instancji $\boldsymbol{x}$ uzyskane przez rozważany algorytm; OPT(x) — optymalne rozwiązanie dla instancji $\boldsymbol{x}$

W sytuacji, gdy nie jesteśmy[2] w stanie zapewnić gwarancji jakości[3] możliwe jest skorzystanie z **algorytmów heurystycznych**. Jakość algorytmów heurystycznych można badać **eskperymentalnie**[4] — sprawdzać wartości funkcji celu dla różnych instancji problemów. Z tego nie można wnioskować żadnej gwarancji, ponieważ nie wiadomo, że będzie ona konsystentnie[5] generować wyniki w ten sposób, ani czy dla innych instancji algorytm będzie zachowywał się podobnie. Można porównywać algorytm heurystyczny z innymi, ale nadal nie wiemy, czy dla wszystkich przypadków występują te zależności i jak dokładne jest to porównanie.
Gdy napotka się problem klasy co najmniej **NP** i nie wiadomo jakim algorytmem heurystycznym powinno się posłużyć, można rozważyć **algorytmy metaheurystyczne**. Algorytmy metaheurystyczne **ograniczają przestrzeń przeszukiwań**, próbując zidentyfikować te regiony o rozwiązaniach o wysokiej jakości, w wyniku zostaje zaoszczędzony czas, ponieważ nie musimy przeszukiwać regionów o rozwiązaniach potencjalnie gorszych jakości [4]. Zaletą algorytmów metaheurystycznych jest to, że nie są rozwiązaniem jednego problemu, lecz obszernej klasy problemów o niewielu wymogach/założeniach wstępnych.

Badany problem to **symetryczny problem komiwojażera**, który polega na znalezieniu najkrótszej ścieżki prowadzącej przez wszystkie wierzchołki grafu dokładnie raz i wracającej

[1]Definicja takiej funkcji jest dowolna, a celem opisywanych algorytmów jest rozwiązanie zadania optymalizacyjnego tej funkcji.

[2]Teoretycznie ϵ zawsze wynosi $<= \infty$ lub $<= k$, gdzie k jest maksimum globalnym funkcji celu (o ile je znamy) tzn. najgorszym rozwiązaniem. Te przypadki zostały odrzucone. Rozważamy sytuację, w której nie można zapewnić spełnienia równości ograniczającej górnie wartość funkcji celu, oprócz tych dwóch.

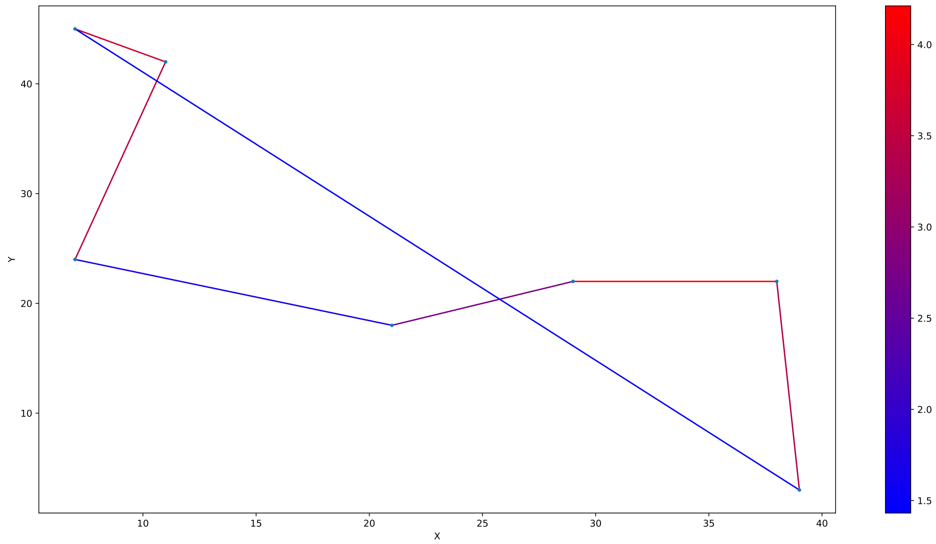
[3]To zazwyczaj wynika z braku znajomości algorytmu rozwiązującego ten problem (co niekoniecznie musi być winą osoby, próbującej rozwiązać problem, być może nikt jeszcze go nie znalazł), lecz istnieją sytuacje, że takie algorytmy po prostu nie istnieją. Takie problemy, w których wykazano, że nie posiadają one algorytmów aproksymacyjnych, nazywa się problemami nieaproksymowalnymi (ang. APX-hard).

[4]Można zaznaczyć na wykresie średnią arytmetyczną i k-krotność odchylenia standardowego. Dalszy opis metodologi użytej do wykonania badań zostanie podany później.

[5]Nie ważne, ile prób podejmiemy, nigdy nie osiągniemy prawdopodobieństwa 100%, że opracowany rozkład prawdopodobieństwa jest prawidłowy. Do tego potrzebny byłby matematyczny dowód, a nie eksperymentalne badania.

do punktu wyjścia[6] w **grafie pełnym**, tak by koszt całego cyklu był jak najmniejszy. Koszt jest sumą wag krawędzi, przez które się przechodzi. Jednak rozważamy taką modyfikację tego problemu, w której kolejność przejść ma znaczenie. Każda instancja problemu będzie posiadać wektor ze skalarami, przez które należy przemnożyć skalarnie wektor wag krawędzi wchodzących w rozwiązanie. W wyniku tego zabiegu, znane algorytmy aproksymacyjne dla STSP nie zachowują swojej gwarancji jakości i użycie metaheurestyk jest uzasadnione.

Dla lepszego przedstawienia problemu, poniżej podano przykładowe rozwiązane instancji tego problemu algorytmem wyczerpującym. Na razie rozpatrywane instancje posiadają grafy, których wagi mogą być opisane za pomocą dystansu na płaszczyźnie euklidesowej, czyli takich, których wierzchołki można umiejscowić w układzie współrzędnych kartezjańskich. Ułatwia to wizualizację wyników i działania algorytmu.



Rysunek 1: Przykładowe rozwiązanie instancji problemu. Większa czerwień krawędzi odpowiada większemu skalarowi; źródło: opracowanie własne

[6]Wierzchołek początkowy/końcowy jest wyjątkiem w tym cyklu i zostaje odwiedzony dwukrotnie.

6 Generowanie grafów

Grafy do wizualizacji:

```
#Generuje graf, ktory mozna ladnie przedstawic graficznie
def generujGrafNaPlaszczyznie(n, punkty, skalary):
    global cache
    graf = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(i):
            graf[i][j] = graf[j][i] = math.sqrt((punkty[j][0] - punkty[i][0])**2 + (punkty[j][1] - punkty[i][1])**2)
            if (graf[i][j] < minDys):
                j = 0
                punkty[i] = (random.randint(0, maxPos), random.randint(0, maxPos))
    cache = [np.array(graf)*skalary[i] for i in range(n)]
    return graf
```

Całkowicie losowe grafy:

```
#Bardziej ogolna metoda generacji grafu
def generujGraf(n, skalary):
    global cache
    graf = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(i):
            graf[i][j] = graf[j][i] = random.random() * maxWaga
    cache = [np.array(graf)*skalary[i] for i in range(n)]
    return graf
```

7 Algorytm wyczerpujący

Algorytmy wyczerpujące sprawdzają wszystkie możliwe rozwiązania, dlatego zawsze osiągają optimum. Możemy sprawdzić poprawność naszego algorytmu, porównując wyniki dla małej liczby wierzchołków.

```
def wyczerpujacy(graf, skalary):
    bestSol = []
    bestSum = sum(max(graf))*max(skalary)
    for p in itertools.permutations([i for i in range(n)]):
        sol = list(p)
        sol.append(sol[0])
        Sum = ocena(sol)
        if bestSum > Sum:
            bestSol = sol
            bestSum = Sum
    return (bestSum, bestSol)
```

8 Algorytm zachłanny

Algorytm zachłanny to taki algorytm, który dokonuje najlepszego lokalnego wyboru tzn. takiego, który przy obecnym zestawie danych jest najkorzystniejszy, gdyby cały algorytm miał zakończyć pracę po tej operacji. Dla opisywanego problemu ten algorytm wygląda tak:

```
def zachlanny(graf, skalary):
    n = len(graf)
    bestSol = []
    bestSum = sum(max(graf))*max(skalary)
    for p in itertools.permutations([i for i in range(n)]):
        sol = list(p)
        sol.append(sol[0])
        Sum = sum([graf[sol[i-1]][sol[i]]*skalary[i-1] for i in range(1, n+1)])
        if bestSum > Sum:
            bestSol = sol
            bestSum = Sum
    return (bestSum, bestSol)
```

9 Algorytm losowy

Dla sprawdzenia, czy metaheurystyka jest poprawnie dobrana dla tego problemu, wyniki porównano z **algorytmem losowym**, ograniczonym czasowo:

```python
def losowy(graf):
    global czas
    start = time.time()
    best = losujSciezke()
    bestsum = ocena(best)
    while time.time()-start<czas:
        sciezka = losujSciezke()
        sum = ocena(sciezka)
        if sum<bestsum:
            best = sciezka
            bestsum = sum
    return sciezka
```

10 Algorytm genetyczny

Algorytmy ewolucyjne to klasa algorytmów metaheurystycznych, które symulują **naturalną ewolucję** [5]. **Algorytm genetyczny** to taki algorytm, który opiera swoje działanie na procesie **mutacji**, **krzyżowania** i **naturalnej selekcji**. Ze zbioru potencjalnych rozwiązań zwanych **populacją** wybiera się osobników za pomocą **operatora selekcji** i na nich dokonuje zmian w rozwiązaniu **operatorem mutacji**, a następnie miesza fragmenty rozwiązań ze sobą **operatorem krzyżowania** [6]. Nowo powstałe rozwiązania są dodane do populacji, lecz jej rozmiar jest **ograniczony**, zatem jedynie najlepsze (wg. funkcji celu) osobniki przetrwają. TSP posiada problem przy krzyżowaniu, ponieważ po takiej operacji z dużym prawdopodobieństwem otrzymane rozwiązania **przestaną być cyklem Hamiltona**, dlatego potrzebne jest odpowiednie krzyżowanie lub procedura naprawy rozwiązania. Istnieje wiele sposobów osiągnięcia tego, nasza grupa wpadła na własną procedurę naprawiania (nie przypisujemy sobie pierwszeństwa wymyślenia tego algorytmu, jest to bardzo mało prawdopodobne. Alternatywne algorytmy krzyżowania to: PMX, CX, OX1, OX2, POS, VS, AP, SCX, SBX, Edge Recombination; wiele z nich według nas ma duże szanse być lepszym niż nasz algorytm):

```python
def napraw(sol, start, koniec):
    global n, indeksy
    wolne = [1] * n                          #wolne[i] posiada wartosc 1, jezeli wierzcholek o indeksie i nie zostal wykorzystany
    for i in range(n):
        wolne[sol[i]] = 0
    random.shuffle(indeksy)                  #indeksy[i] to losowy wierzcholek, po to by naprawa wrzucala losowy wierzcholek w miejsce nieprawidlowego
    k = 0
    uzyte = [0] * n                          #uzyte sluzy do znalezienia powtarzajacych sie wierzcholkow
    for i in range(start, koniec):           #nie zmieniamy wierzcholkow z fragmentu, ktory zostal wrzucony z krzyzowania, dlatego te wszystkie wierzcholki
        uzyte[sol[i]] = 1                    #sa od razu w uzyciu i to one wymuszaja poprawe wierzcholka, gdy taki sam wystepuje w otoczeniu tego fragmentu
    for i in range(n):
        if i >= start and i < koniec:
            continue
        if uzyte[sol[i]]:
            while (not wolne[indeksy[k]]):
                k += 1
            sol[i] = indeksy[k]
            wolne[indeksy[k]] = 0
            uzyte[sol[i]] = 1
    sol[n]=sol[0]
```

Wykorzystano kilka struktur optymalizacyjnych, dlatego zmiany parametrów algorytmu genetycznego wymagają ich regeneracji, w związku z czym stworzono pomocniczą procedurę:

```
def kalibrujAlgorytm(newN, newPopulacja = populacja, newGeneracje = generacje):
    global n, populacja, generacje, indeksy, kolejnosc, wstepnaPopulacja, stop
    n = newN                                                #liczba wierzcholkow
    populacja = newPopulacja
    generacje = newGeneracje
    wstepnaPopulacja = populacja*4
    #sekcja optymalizacyjna
    indeksy = [i for i in range(n)]
    kolejnosc = [i for i in range(populacja)]
    stop = populacja - (populacja%2)
```

Pełny algorytm wraz z operatorami i funkcjami pomocniczymi:

```
#Generuje rozwiazania, z ktorych powstaje wstepna populacja
def losujSciezke():
    global indeksy
    solution = copy.deepcopy(indeksy)
    random.shuffle(solution)
    solution.append(solution[0])
    return solution

#Funkcja oceny
def ocena(sol):
    global cache
    global n
    suma = 0
    for i in range(n):
        #if (sol[i] == sol[i+1]):
            #print(sol)                                 #nie powinno sie zdarzyc i sie nie zdarza. Odkomentowac dla testow
        suma += cache[i][sol[i]][sol[i+1]]              #cache to tablica 3 wymiarowa, ktora zawiera wszystkie przeskalowane
                                                        #wagi krawedzi, zeby nie trzeba bylo tego robic za kazdym razem
    return suma

def napraw(sol, start, koniec):
    global n, indeksy
    wolne = [1] * n                                     #wolne[i] posiada wartosc 1, jezeli wierzcholek o indeksie i nie zostal wykorzystany
    for i in range(n):
        wolne[sol[i]] = 0
    random.shuffle(indeksy)                             #indeksy[i] to losowy wierzcholek, po to by naprawa wrzucala losowy wierzcholek w miejsce nieprawidlowego
    k = 0
    uzyte = [0] * n                                     #uzyte sluzy do znalezienia powtarzajacych sie wierzcholkow
    for i in range(start, koniec):                      #nie zmieniamy wierzcholkow z fragmentu, ktory zostal wrzucony z krzyzowania, dlatego te wszystkie wierzcholki
        uzyte[sol[i]] = 1                               #sa od razu w uzyciu i to one wymuszaja poprawe wierzcholka, gdy taki sam wystepuje w otoczeniu tego fragmentu
    for i in range(n):
        if i >= start and i < koniec:
            continue
        if uzyte[sol[i]]:
            while (not wolne[indeksy[k]]):
                k += 1
            sol[i] = indeksy[k]
            wolne[indeksy[k]] = 0
        uzyte[sol[i]] = 1
    sol[n]=sol[0]

#Wybranie z populacji "najlepszych" osobnikow
def operatorSelekcji(sols):
    global populacja
    newsols = sols
    [newsols.append(x) for x in sols if x not in newsols]
    sols = newsols
    sols.sort(key=ocena)
    sols=sols[:populacja]

#Drobne losowe zmiany
def operatorMutacji(sol):
    global n
    w1 = random.randint(0, n - 1)                       #losowanie dwoch wierzcholkow na sciezce
    w2 = random.randint(1, n - 1)
    if w2 <= w1:
        w2 = w2 - 1                                     #dzieki temu zabiegowi mamy gwarancje, ze w1 != w2
    sol[w1], sol[w2] = sol[w2], sol[w1]                 #zamiana miejscami dwoch wybranych wierzcholkow
    sol[n]=sol[0]                                       #dla pewnosci, jezeli w1 albo w2 bylo rowne 0, trzeba poprawic rozwiazanie

#Stworzenie dwoch nowych rozwiazan na podstawie dwoch podanych
def krzyzuj(sols, indeks1, indeks2):
    sol1 = sols[indeks1]
    sol2 = sols[indeks2]
    start = random.randint(0,n-2)
    koniec = random.randint(start,n-2)
    newSol1 = sol1[:start]+sol2[start:koniec]+sol1[koniec:] #utworzenie nowego osobnika nakladajac wylosowany fragment na pierwszego osobnika
    napraw(newSol1, start, koniec)                      #doprowadzenie nowego osobnika do stanu w ktorym przestawia on cykl Hamiltona
    operatorMutacji(newSol1)                            #mozna zamienic kolejnosc krzyzowania z mutacja, ale z naszych testow wynika, ze nie ma to duzego wplywu
    newSol2 = sol2[:start]+sol1[start:koniec]+sol2[koniec:]
    napraw(newSol2, start, koniec)
    operatorMutacji(newSol2)
    return (newSol1,newSol2)
```

```
#Zmieszanie dwoch rozwiazan w celu poszukiwania nowych
def operatorKrzyzowania(sols):
    global kolejnosc, populacja, stop
    random.shuffle(kolejnosc)
    j = 0
    while (j != stop):
        indeks1 = kolejnosc[j]
        j += 1
        indeks2 = kolejnosc[j]
        j += 1
        (newSol1, newSol2) = krzyzuj(sols, indeks1, indeks2)
        sols.append(newSol1)
        sols.append(newSol2)

#Wykonuje algorytm genetyczny
def genetyczny(graf):
    global wstepnaPopulacja, populacja, generacje
    sols = [losujSciezke() for _ in range(wstepnaPopulacja)] #wylosowanie startowej populacji
    sols.append(zachlanny(graf))

    sols.sort(key=ocena)
    operatorSelekcji(sols)
    for i in range(generacje):
        operatorKrzyzowania(sols)
        operatorSelekcji(sols)
    return sols[0]
```

10.1 Wizualizacja pracy algorytmu

Każdy z poniższych przykładów znajduje lepsze rozwiązanie 15 razy i zakończa swoją pracę.

- Liczba wierzchołków 20; populacja 20; populacja wstępna zawiera wynik algorytmu zachłannego
- Liczba wierzchołków 100; populacja 20; populacja wstępna zupełnie losowa
- Liczba wierzchołków 60; populacja 10; populacja wstępna zupełnie losowa
- Liczba wierzchołków 60; populacja 50; populacja wstępna zupełnie losowa
- Liczba wierzchołków 60; populacja 100; populacja wstępna zupełnie losowa
- Liczba wierzchołków 100; populacja 20; populacja wstępna zawiera wynik algorytmu zachłannego
- Liczba wierzchołków 100; populacja 80; populacja wstępna zawiera wynik algorytmu zachłannego
- Liczba wierzchołków 160; populacja 40; populacja wstępna zawiera wynik algorytmu zachłannego

11 Testy

11.1 Cechy

Badanymi cechami algorytmu będzie **czas wykonania**, **jakość** rozwiązań w zależności od **liczby wierzchołków**. Zostaną narysowane różne przebiegi algorytmu o różnym parametrze $\boldsymbol{m}$ — **maksymalnej populacji**, $\boldsymbol{k}$ — **liczby krzyżowań**.

11.2 Metodologia

Testy są generowane dla grafów całkowicie losowych, metodą opisaną w rozdziale 6. Wartości skalarów znajdują się w przedziale $\langle 1.4, 4.4\rangle$. Maksymalna waga krawędzi wynosi 50. Wstępna populacja wygenerowana losowo wynosi 4*populacja (parametr algorytmu). Dla wszystkich badanych cech obliczono odchylenie standardowe ze wzoru:

$$\sigma_N = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{4}$$

We wzorze pojawia się $N-1$ zamiast N, by zrekompensować błąd systematyczny. Liczba pomiarów wynosi nie mniej niż 10, dla każdego kolejnego N obliczamy odchylenie standardowe. Można wziąć średnią z tych odchyleń i obliczyć odchylenie standardowe wartości średniej ze wzoru:

$$\sigma_\sigma = \sqrt{\frac{1}{N\,(N-1)}\sum_{i=1}^{N}(\sigma_i - \bar{\sigma})^2} \tag{5}$$

Te „odchylenie odchyleń” mówi o niepewności „rzeczywistego odchylenia”:

$$\lim_{n\to\infty} \sigma_n \tag{6}$$

Program dobierze taką próbkę n, że σ_σ nie powinno przekroczyć 5% $\bar{\sigma}$. Z kolei na mocy **nierówności Czybyszewa** z prawdopodobieństwem $\frac{24}{25} = 96\%$ zachodzi:

$$\bar{\sigma} \pm 5\sigma_\sigma \in \lim_{n\to\infty} \sigma_n \implies \bar{\sigma} \pm 25\% \in \lim_{n\to\infty} \sigma_n$$

Zatem $\sigma_{FINAL} = 125\%\bar{\sigma}$. Jeżeli skorzystać ponownie z nierówności Czybyszewa, można stwierdzić, że z prawdopodobieństwem $\frac{8}{9} \approx 89\%$ zachodzi:

$$|x_i - \bar{x}| \leqslant 3\sigma_{FINAL}$$

Ostateczny wniosek:
Z prawdopodobieństwem $\frac{24\times 8}{25\times 9} \approx 85\%$ losowe rozwiązanie będzie posiadać cechę:

$$x_i \leqslant \bar{x} + 3\sigma_{FINAL}$$

Komentarz:
Przedmiot „Statystyka i analiza danych” jest dopiero na następnym semestrze. Mamy szczerą nadzieję, że powyższe rozumowanie jest poprawnie i jednocześnie jesteśmy niemal pewni, że istniał o wiele lepszy sposób na sporządzenie tych danych.

11.3 Pełny program

```
import random
import time
import copy
import math
import itertools
import statistics

import matplotlib
import matplotlib.pyplot as plot
import numpy as np

n = 800                                         #liczba wierzcholkow
populacja = 10                                  #licznosc populacji
wstepnaPopulacja = populacja*4                  #liczba losowan rozwiazan do wygenerowania wstepnej populacji
generacje = 20                                  #liczba generacji
maxPos = 50                                     #maksymalne x i y wierzcholka (musi byc wieksze od 1) w pierwszym sposobie generowania grafu
minDys = 2                                      #minimalny dystans w 1. metodzie generacji algorytmu
maxWaga = 50                                    #maksymalna waga krawedzi w drugim sposobie generowania grafu
minSkalar = 1.4                                 #minimalna wartosc skalara (zmiennoprzecikowa)
maxSkalar = 4.4                                 #maksymalna wartosc skalara (zmiennoprzecikowa)
skalaWykresu = 8                                #zmienna wprowadzajace kosmetyczne zmiany
powtorzenia = 100                               #powtorzenia algorytmu genetycznego dla jednego grafu
czas = 2                                        #maksymalny czas w sekundach dzialania algorytmu losowego
#sekcja optymalizacyjna
cache = []
indeksy = [i for i in range(n)]
kolejnosc = [i for i in range(populacja)]
stop = populacja - (populacja%2)

#Generuje graf, ktory mozna ladnie przedstawic graficznie
def generujGrafNaPlaszczyznie(n, punkty, skalary):
    global cache
    graf = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(i):
            graf[i][j] = graf[j][i] = math.sqrt((punkty[j][0] - punkty[i][0])**2 + (punkty[j][1] - punkty[i][1])**2)
            if (graf[i][j] < minDys):
                j = 0
                punkty[i] = (random.randint(0, maxPos), random.randint(0, maxPos))
    cache = [np.array(graf)*skalary[i] for i in range(n)]
    return graf

#Bardziej ogolna metoda generacji grafu
def generujGraf(n, skalary):
    global cache
    graf = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(i):
            graf[i][j] = graf[j][i] = random.random() * maxWaga
    cache = [np.array(graf)*skalary[i] for i in range(n)]
    return graf

#Generuje rozwiazania, z ktorych powstaje wstepna populacja
def losujSciezke():
    global indeksy
    solution = copy.deepcopy(indeksy)
    random.shuffle(solution)
    solution.append(solution[0])
    return solution

#Funkcja oceny
def ocena(sol):
    global cache
    global n
    suma = 0
    for i in range(n):
        #if (sol[i] == sol[i+1]):
            #print(sol)                         #nie powinno sie zdarzyc i sie nie zdarza. Odkomentowac dla testow
        suma += cache[i][sol[i]][sol[i+1]]      #cache to tablica 3 wymiarowa, ktora zawiera wszystkie przeskalowane
                                                #wagi krawedzi, zeby nie trzeba bylo tego robic za kazdym razem
    return suma

def napraw(sol, start, koniec):
    global n, indeksy
    wolne = [1] * n                             #wolne[i] posiada wartosc 1, jezeli wierzcholek o indeksie i nie zostal wykorzystany
    for i in range(n):
        wolne[sol[i]] = 0
    random.shuffle(indeksy)                     #indeksy[i] to losowy wierzcholek, po to by naprawa wrzucala losowy wierzcholek w miejsce
        nieprawidlowego
    k = 0
    uzyte = [0] * n                             #uzyte sluzy do znalezienia powtarzajacych sie wierzcholkow
    for i in range(start, koniec):              #nie zmieniamy wierzcholkow z fragmentu, ktory zostal wrzucony z krzyzowania, dlatego te
        wszystkie wierzcholki
        uzyte[sol[i]] = 1                       #sa od razu w uzyciu i to one wymuszaja poprawe wierzcholka, gdy taki sam wystepuje w otoczeniu
            tego fragmentu
    for i in range(n):
        if i >= start and i < koniec:
            continue
        if uzyte[sol[i]]:
            while (not wolne[indeksy[k]]):
                k += 1
            sol[i] = indeksy[k]
```

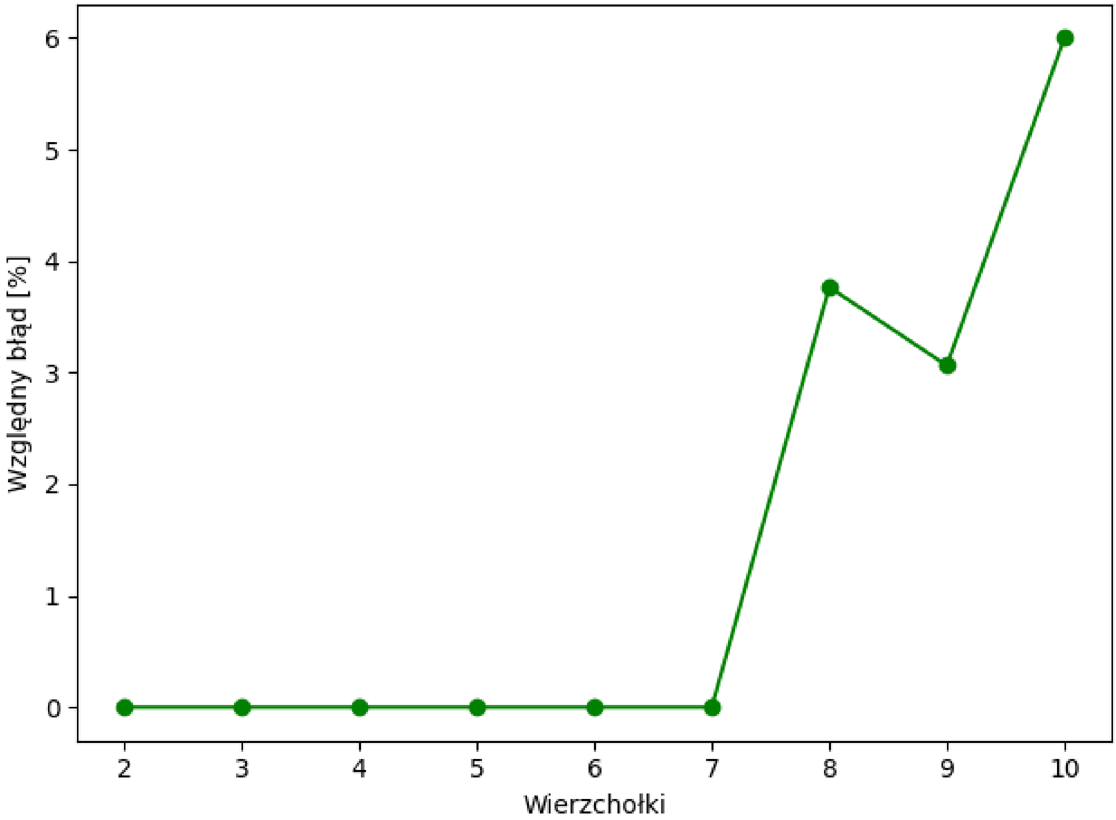
```python
            wolne[indeksy[k]] = 0
        uzyte[sol[i]] = 1
    sol[n]=sol[0]

#Wybranie z populacji "najlepszych" osobnikow
def operatorSelekcji(sols):
    global populacja
    newsols = sols
    [newsols.append(x) for x in sols if x not in newsols]
    newsols.sort(key=ocena)
    return newsols[:populacja]

#Drobne losowe zmiany
def operatorMutacji(sol):
    global n
    w1 = random.randint(0, n - 1)                           #losowanie dwoch wierzcholkow na sciezce
    w2 = random.randint(1, n - 1)
    if w2 <= w1:
        w2 = w2 - 1                                         #dzieki temu zabiegowi mamy gwarancje, ze w1 != w2
    sol[w1], sol[w2] = sol[w2], sol[w1]                     #zamiana miejscami dwoch wybranych wierzcholkow
    sol[n]=sol[0]                                           #dla pewnosci, jezeli w1 albo w2 bylo rowne 0, trzeba poprawic rozwiazanie

#Stworzenie dwoch nowych rozwiazan na podstawie dwoch podanych
def krzyzuj(sols, indeks1, indeks2):
    sol1 = sols[indeks1]
    sol2 = sols[indeks2]
    start = random.randint(0,n-2)
    koniec = random.randint(start,n-2)
    newSol1 = sol1[:start]+sol2[start:koniec]+sol1[koniec:] #utworzenie nowego osobnika nakladajac wylosowany fragment na pierwszego osobnika
    napraw(newSol1, start, koniec)                          #doprowadzenie nowego osobnika do stanu w ktorym przestawia on cykl Hamiltona
    operatorMutacji(newSol1)                                #mozna zamienic kolejnosc krzyzowania z mutacja, ale z naszych testow wynika, ze nie ma to duzego wplywu
    newSol2 = sol2[:start]+sol1[start:koniec]+sol2[koniec:]
    napraw(newSol2, start, koniec)
    operatorMutacji(newSol2)
    return (newSol1,newSol2)

#Zmieszanie dwoch rozwiazan w celu poszukiwania nowych
def operatorKrzyzowania(sols):
    global kolejnosc, populacja, stop
    random.shuffle(kolejnosc)
    j = 0
    while (j != stop):
        indeks1 = kolejnosc[j]
        j += 1
        indeks2 = kolejnosc[j]
        j += 1
        (newSol1, newSol2) = krzyzuj(sols, indeks1, indeks2)
        sols.append(newSol1)
        sols.append(newSol2)

#Wykonuje algorytm genetyczny
def genetyczny(graf):
    global wstepnaPopulacja, populacja, generacje
    sols = [losujSciezke() for _ in range(wstepnaPopulacja)]
    sols.append(zachlanny(graf))

    sols = operatorSelekcji(sols)
    for i in range(generacje):
        operatorKrzyzowania(sols)
        sols = operatorSelekcji(sols)
    return sols[0]

#Ustawia nowe parametry algorytmu i generuje struktury pomocnicze (optymalizacyjne)
def kalibrujAlgorytm(newN, newPopulacja = populacja, newGeneracje = generacje):
    global n, populacja, generacje, indeksy, kolejnosc, wstepnaPopulacja, stop
    n = newN                                               #liczba wierzcholkow
    populacja = newPopulacja
    generacje = newGeneracje
    wstepnaPopulacja = populacja*4
    #sekcja optymalizacyjna
    indeksy = [i for i in range(n)]
    kolejnosc = [i for i in range(populacja)]
    stop = populacja - (populacja%2)

#Algorytm wyczerpujacy dla naszego problemu
def wyczerpujacy(graf, skalary):
    bestSol = []
    bestSum = n * max(max(graf))*max(skalary)
    for p in itertools.permutations([i for i in range(n)]):
        sol = list(p)
        sol.append(sol[0])
        Sum = ocena(sol)
        if bestSum > Sum:
            bestSol = sol
            bestSum = Sum
    return (bestSum, bestSol)

def zachlanny(graf):
    dowybrania = [i for i in range(1,n)]
    sol = [0]
    wybrane=0
    for i in range(0,n-1):
        best = dowybrania[0]
        for w in dowybrania:
```

```
            if cache[wybrane][sol[wybrane]][best]>cache[wybrane][sol[wybrane]][w]:
                best = w
        dowybrania.remove(best)
        sol.append(best)
        wybrane=wybrane+1
    sol.append(sol[0])
    return sol

def losowy(graf):
    global czas
    start = time.time()
    best = losujSciezke()
    bestsum = ocena(best)
    while time.time()-start<czas:
        sciezka = losujSciezke()
        sum = ocena(sciezka)
        if sum<bestsum:
            best = sciezka
            bestsum = sum
    return sciezka

def testuj(algorytm, graf):
    global maxSkalar, minSkalar, maxPos, n
    wyniki = []
    czasy = []
    odchyleniaWynik = []
    odchyleniaCzas = []
    #print("0")
    start = time.time()
    wyniki.append(ocena(algorytm(graf)))
    czasy.append(time.time() - start)
    for i in range(1, 10):
    #    print(i)
        start = time.time()
        wyniki.append(ocena(algorytm(graf)))
        czasy.append(time.time() - start)
        odchyleniaWynik.append(statistics.stdev(wyniki))
        odchyleniaCzas.append(statistics.stdev(czasy))

    if (statistics.fmean(odchyleniaWynik) != 0):
        while (statistics.stdev(odchyleniaWynik) / math.sqrt(len(odchyleniaWynik)) > statistics.fmean(
                odchyleniaWynik) / 20 or
               (statistics.fmean(odchyleniaCzas) != 0 and statistics.stdev(odchyleniaCzas) / math.sqrt(
                   len(odchyleniaCzas)) > statistics.fmean(odchyleniaCzas) / 20)):
            #print("ooj: ", statistics.stdev(odchyleniaWynik) / math.sqrt(len(odchyleniaWynik)) / statistics.fmean(
            #    odchyleniaWynik) * 100)
            #if (statistics.fmean(odchyleniaCzas) != 0):
            #    print("oocz: ", statistics.stdev(odchyleniaCzas) / math.sqrt(len(odchyleniaCzas)) / statistics.fmean(
            #        odchyleniaCzas) * 100)
            start = time.time()
            wyniki.append(ocena(algorytm(graf)))
            czasy.append(time.time() - start)
            odchyleniaWynik.append(statistics.stdev(wyniki))
            odchyleniaCzas.append(statistics.stdev(czasy))

    #print("Srednia ocena jakosci cyklu: ", statistics.fmean(wyniki))
    #print("Odchylenie odchylen jakosci: ", statistics.stdev(odchyleniaWynik)/math.sqrt(len(odchyleniaWynik)))
    #print("Srednie odchylenie jakosci: ", statistics.fmean(odchyleniaWynik))
    #print("Srednia ocena czasu cyklu: ", statistics.fmean(czasy))
    #print("Odchylenie odchylen czasu: ", statistics.stdev(odchyleniaCzas)/math.sqrt(len(odchyleniaCzas)))
    #print("Srednie odchylenie czasu: ", statistics.fmean(odchyleniaCzas))
    bladWOsiYJakosc = 15 * statistics.fmean(odchyleniaWynik)/4
    bladWOsiYCzas = 15 * statistics.fmean(odchyleniaCzas)/4
    return (statistics.fmean(wyniki), statistics.fmean(czasy), bladWOsiYJakosc, bladWOsiYCzas)

def rysujKrawedz(i, rozwiazanie, punkty, skalary, skalaWykresu):
    indeks = rozwiazanie[i]
    indeksPrev = rozwiazanie[i-1]
    #Kolor zalezy od wartosci skalara. Minimalna wartosc jest niebieska, maksymalna czerwona, reszta pomiedzy
    if (np.unique(max(skalary))-np.unique(min(skalary)) == 0):
        kolor = 0
    else:
        kolor = (skalary[i-1]-np.unique(min(skalary)))/(np.unique(max(skalary))-np.unique(min(skalary)))
    plot.plot([punkty[indeksPrev][0], punkty[indeks][0]], [punkty[indeksPrev][1], punkty[indeks][1]],
              c = (kolor[0],0,1-kolor[0]), markersize = skalaWykresu/4, zorder=1)

def rysujRozwiazanie(rozwiazanie, title = "", skalaWykresu = skalaWykresu):
    plot.scatter([punkty[i][0] for i in range(n)], [punkty[i][1] for i in range(n)], s=skalaWykresu, zorder=2)
    plot.title(title)
    plot.xlabel("X")
    plot.ylabel("Y")
    for i in range(1, len(rozwiazanie)):
        rysujKrawedz(i, rozwiazanie, punkty, skalary, skalaWykresu)
    N = 256
    vals = np.ones((N, 4))
    vals[:, 0] = np.linspace(0, 1, N)
    vals[:, 1] = np.linspace(0, 0, N)
    vals[:, 2] = np.linspace(1, 0, N)
    newcmp = matplotlib.colors.ListedColormap(vals)
    plot.colorbar(matplotlib.cm.ScalarMappable(norm=matplotlib.colors.Normalize(vmin=min(skalary), vmax=max(skalary)),cmap=newcmp))

def appendfile(filename,tekst):
    f = open(filename,"at")
    f.write(tekst)
    f.close
```

```
pre_file_name =time.strftime("%d_%m_%y_%H_%M_%S")
naglowki = ["n","czas1020","czas1050","czas2020","czas2050","czas4020","czas4040","czassredni","czaslosowy","czaszachlanny",
            "jak1020","jak1050","jak2020","jak2050","jak4020","jak4040","jaklosowy","jakzachlanny",
            "bczas1020","bczas1050","bczas2020","bczas2050","bczas4020","bczas4040","bczaslosowy","bczaszachlanny",
            "bjak1020","bjak1050","bjak2020","bjak2050","bjak4020","bjak4040","bjaklosowy","bjakzachlanny"]

appendfile(pre_file_name,' '.join(naglowki)+"\n")
dane = [1,50]+[i for i in range(100,600,100)]
#dane = [i for i in range(11)]
#dane = [i for i in range(3, 11)] #na razie bez tych wiekszych danych
tablicaJakosci = [[] for _ in range(len(dane))]
tablicaCzasu = [[] for _ in range(len(dane))]
tablicaBleduJakosci = [[] for _ in range(len(dane))]
tablicaBleduCzasu = [[] for _ in range(len(dane))]


for i in range(len(dane)-1, -1,-1): #zaczynam od konca, zeby nie sprawdzic czy algorytm trwa za dlugo
    n = dane[i]
                                                                                                #generacja nowego grafu
    skalary = [(minSkalar + random.random() * (maxSkalar - minSkalar)) for _ in range(n)]
    #punkty = [(random.randint(0, maxPos), random.randint(0, maxPos)) for _ in range(n)]
    graf = generujGraf(n, skalary)
    start = time.time()


    #t = time.time()
    #(sum,a)=wyczerpujacy(graf,skalary)
    #tk = time.time()
    #print(n," ",sum," ",tk-t)

    kalibrujAlgorytm(n, 10, 20)
    (jakosc, czas, bladWOsiYJakosc, bladWOsiYCzas) = testuj(genetyczny, graf)
    tablicaJakosci[i].append(jakosc)
    tablicaCzasu[i].append(czas)
    tablicaBleduJakosci[i].append(bladWOsiYJakosc)
    tablicaBleduCzasu[i].append(bladWOsiYCzas)
    kalibrujAlgorytm(n, 10, 50)
    (jakosc, czas, bladWOsiYJakosc, bladWOsiYCzas) = testuj(genetyczny, graf)
    tablicaJakosci[i].append(jakosc)
    tablicaCzasu[i].append(czas)
    tablicaBleduJakosci[i].append(bladWOsiYJakosc)
    tablicaBleduCzasu[i].append(bladWOsiYCzas)
    kalibrujAlgorytm(n, 20, 20)
    (jakosc, czas, bladWOsiYJakosc, bladWOsiYCzas) = testuj(genetyczny, graf)
    tablicaJakosci[i].append(jakosc)
    tablicaCzasu[i].append(czas)
    tablicaBleduJakosci[i].append(bladWOsiYJakosc)
    tablicaBleduCzasu[i].append(bladWOsiYCzas)
    kalibrujAlgorytm(n, 20, 50)
    (jakosc, czas, bladWOsiYJakosc, bladWOsiYCzas) = testuj(genetyczny, graf)
    tablicaJakosci[i].append(jakosc)
    tablicaCzasu[i].append(czas)
    tablicaBleduJakosci[i].append(bladWOsiYJakosc)
    tablicaBleduCzasu[i].append(bladWOsiYCzas)
    kalibrujAlgorytm(n, 40, 20)
    (jakosc, czas, bladWOsiYJakosc, bladWOsiYCzas) = testuj(genetyczny, graf)
    tablicaJakosci[i].append(jakosc)
    tablicaCzasu[i].append(czas)
    tablicaBleduJakosci[i].append(bladWOsiYJakosc)
    tablicaBleduCzasu[i].append(bladWOsiYCzas)
    kalibrujAlgorytm(n, 40, 40)
    (jakosc, czas, bladWOsiYJakosc, bladWOsiYCzas) = testuj(genetyczny, graf)
    tablicaJakosci[i].append(jakosc)
    tablicaCzasu[i].append(czas)
    tablicaBleduJakosci[i].append(bladWOsiYJakosc)
    tablicaBleduCzasu[i].append(bladWOsiYCzas)

    #czas=sum(tablicaCzasu[i])/len(tablicaCzasu[i])
    tablicaCzasu[i].append(czas)

    (jakosc, czas, bladWOsiYJakosc, bladWOsiYCzas) = testuj(losowy, graf)
    tablicaJakosci[i].append(jakosc)
    tablicaCzasu[i].append(czas)
    czas1=czas
    tablicaBleduJakosci[i].append(bladWOsiYJakosc)
    tablicaBleduCzasu[i].append(bladWOsiYCzas)
    #tak zeby losowy mial jakis sensowny czas wykonania
    (jakosc, czas, bladWOsiYJakosc, bladWOsiYCzas) = testuj(zachlanny, graf)
    tablicaJakosci[i].append(jakosc)
    tablicaCzasu[i].append(czas)
    tablicaBleduJakosci[i].append(bladWOsiYJakosc)
    tablicaBleduCzasu[i].append(bladWOsiYCzas)
    appendfile(pre_file_name,str(n)+" "+' '.join([str(n) for n in tablicaCzasu[i]])+" "+' '.join([str(n) for n in tablicaJakosci[i]])+" "+' '.join([str(n) for n in tablicaBleduCzasu[i]])+" "+' '.join([str(n) for n in tablicaBleduJakosci[i]])+"\n")
    #zapisz dane, zeby potem mozna uzyc w wykresie
    #Bledy zapisz po prostu w tablicy i przekaz tablice potem do yerr
    print(time.time() - start)
```

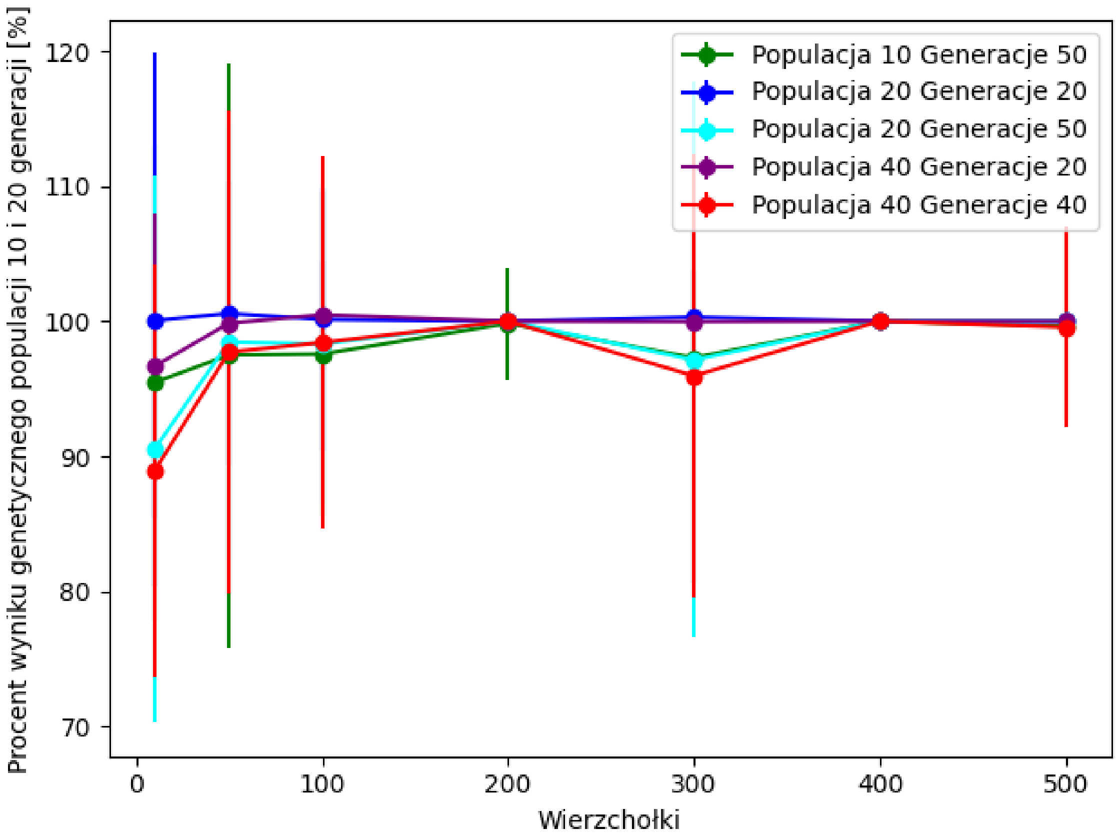
12 Wyniki

Na początku nasza grupa przetestowała względną poprawność algorytmu genetycznego w stosunku do algorytmu wyczerpującego:

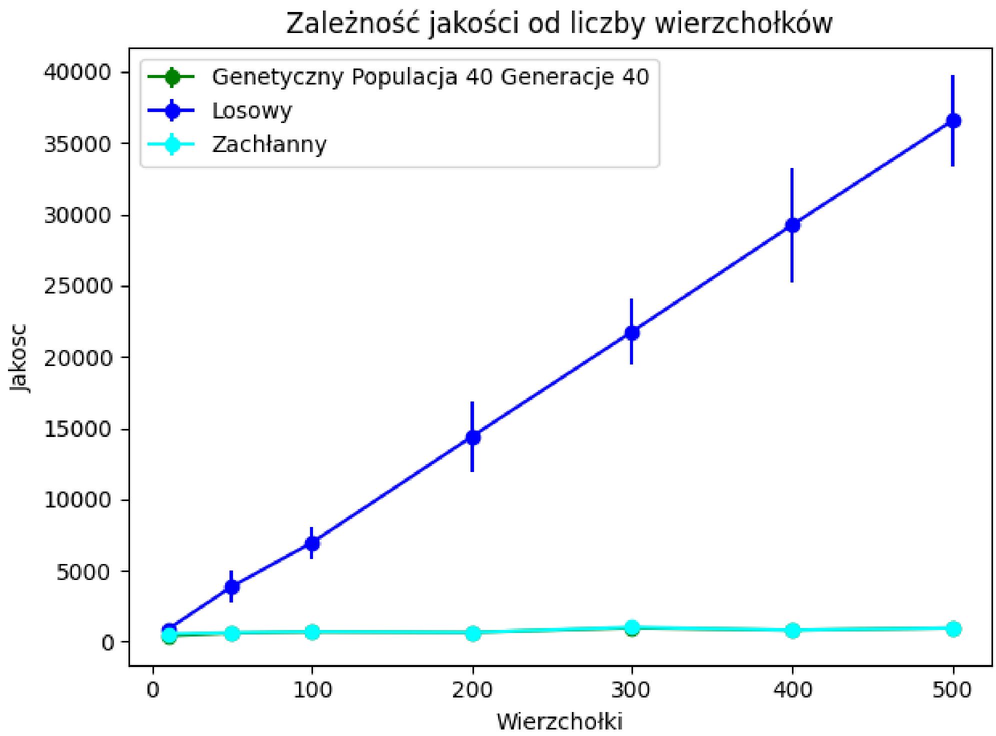


Rysunek 2: Poprawność algorytmu genetycznego w stosunku do algorytmu wyczerpującego; źródło: opracowanie własne

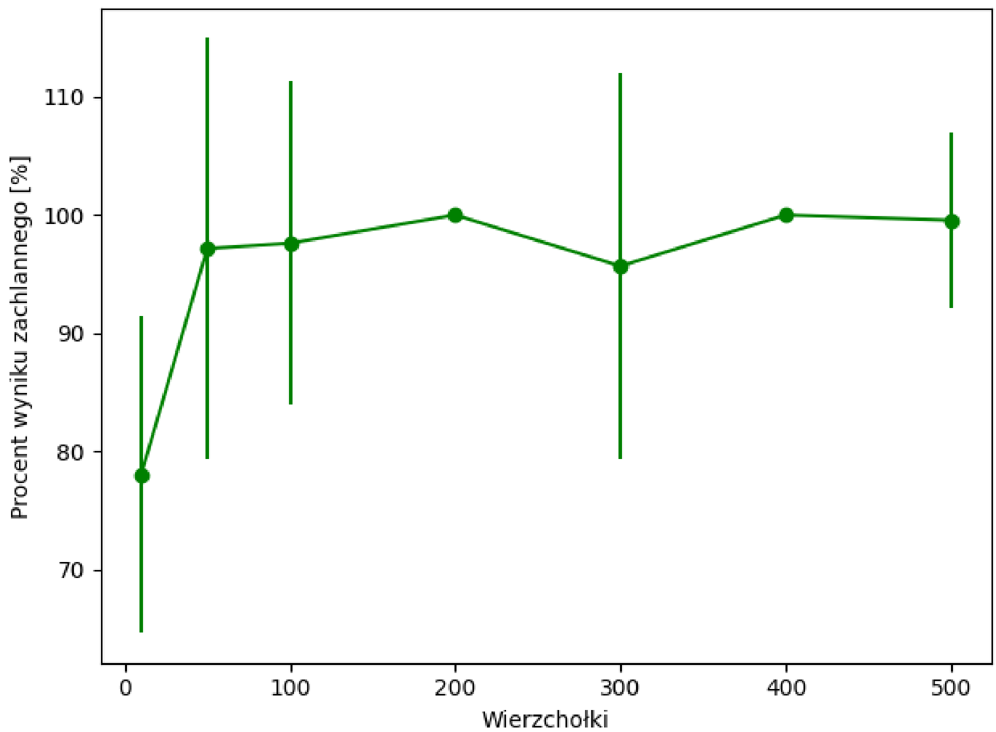
Porównanie jakości rozwiązań (względem najgorzej wypadającego algorytmu z zestawienia):



Rysunek 3: Porównanie jakości algorytmów genetycznych. Najgorzej wypadł algorytm o populacji 10 i generacji 20; źródło: opracowanie własne

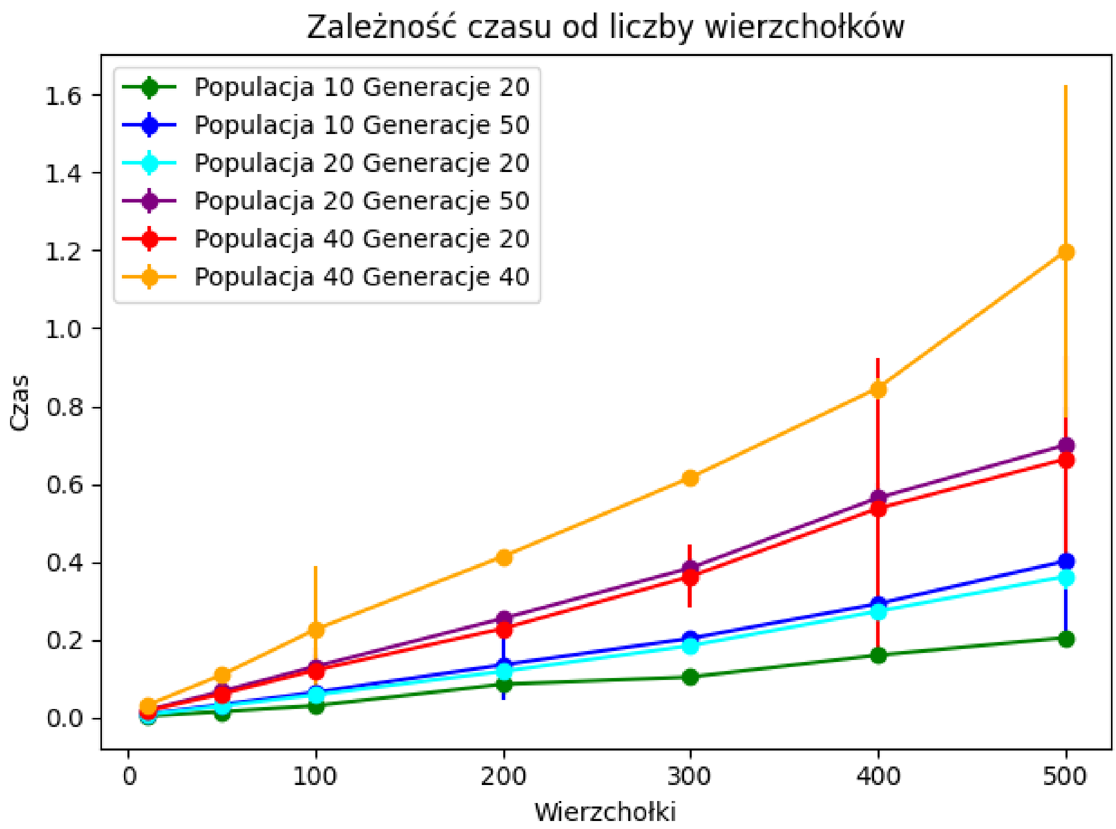


Rysunek 4: Algorytm zupełnie losowy jest bardzo nieefektywny; źródło: opracowanie własne

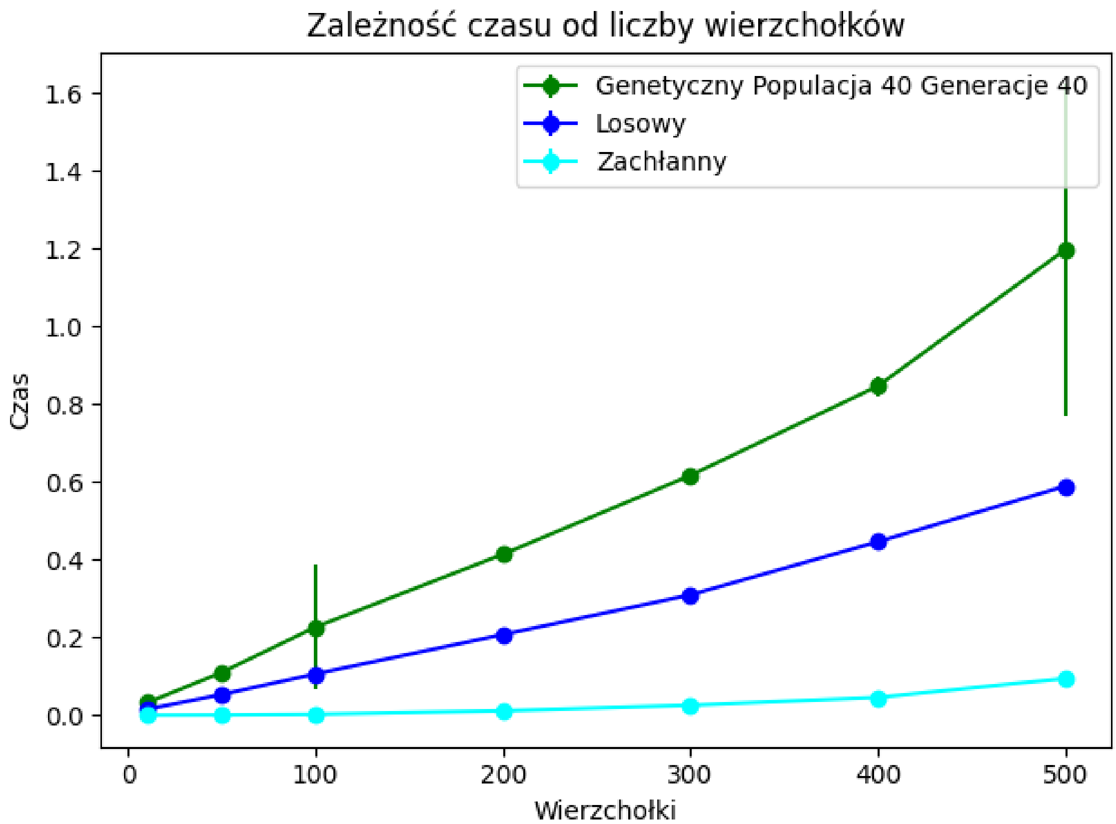


Rysunek 5: Porównanie wyników algorytmu genetycznego o populacji 40 i generacji 40 do zachłannego; źródło: opracowanie własne

Porównanie czasu wykonywania:



Rysunek 6: Porównanie czasu algorytmów genetycznych; źródło: opracowanie własne



Rysunek 7: Porównanie czasu algorytmu genetycznego o populacji 40 i generacji 40 do pozostałych; źródło: opracowanie własne

13 Wnioski

Algorytmy metaheurystyczne mogą być zastosowane do rozwiązywania nieznanych nam przedtem problemów. Jeżeli problem nie jest całkowicie losowy, takie algorytmy wykorzystają zależności wiążące dane i wypracują rozwiązania lepsze niżeli algorytm zupełnie losowy. Zbudowanie dobrego algorytmu metaheurystycznego jest trudne, ponieważ istnieje wiele kombinacji parametrów i sposobów definicji niektórych z jego elementów (w tym przypadku np. operatorów).

13.1 Analiza złożności

Pesymistyczna złożoność obliczeniowa wynosi:

$$O\left(n^2\right)$$

Lecz zależy ona od dwóch istotnych parametrów algorytmu, więc w nieformalnym zapisie ta złożoność wynosi:

$$O\left(generacje \times populacja \times n^2\right)$$

Wąskie gardło to *genetyczny(graf)* → *operatorKrzyzowania(sols)* → *krzyzuj(sols, indeks1, indeks2)* → *napraw(sol, start, koniec)*.
Pesymistyczna złożoność pamięciowa wynosi:

$$O\left(n^3\right)$$

Wąskie gardło to optymalizacyjna struktura pomocnicza **cache**.

14 Możliwe dalsze ścieżki rozwoju projektu (iteracyjnie)

- Porównanie innych operatorów selekcji, krzyżowania (m.in. te wymienione w rozdziale 10) i mutacji
- Dokładniejsze badania wpływu parametrów algorytmu na jego działanie
- Szukanie instancji problemu trudnych dla naszego algorytm
- Zachowanie algorytmu, gdy generacja skalarów i/lub wag jest inna (wg. jakiegoś ciągu lub innych generatorów losowych)
- Porównanie z innymi algorytmami metaheurystycznymi (dla problemu TSP rozsądne wydaje się m.in.: tabu search i algorytm mrówkowy)
- Algorytm wyboru metaheurystyki w zależności od zadanych parametrów (np. wymaganego współczynnika jakości/czasu)
- Badanie współbieżnej metaheurystyki
- Porównanie z hiper-heurystykami
- Zastosowania hiper/metaheurystyk w uczeniu maszynowym

15 Literatura

References

[1] D. P. Williamson and D. B. Shmoys, "The Design of Approximation Algorithms," in. Cambridge: Cambridge University Press, 2010, ch. 1, pp. 13–28, dostęp: 09.01.2021.

[2] V. Kann, "On the Approximability of NP-complete Optimization Problems," in. Stockholm: Royal Institute of Technology, 1992, ch. 1, pp. 8–11, dostęp: 10.01.2021.

[3] R. L. Rardin and R. Uzsoy, "Experimental Evaluation of Heuristic Optimization Algorithms: A Tutorial," *Journal of Heuristics*, vol. 7, pp. 261–304, 2001.

[4] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr, "A survey on metaheuristics for stochastic combinatorial optimization," in. 2009, vol. 8, pp. 239–287, dostęp: 10.02.2021.

[5] S. Bhargava, "A Note on Evolutionary Algorithms and Its Applications," *Adults Learning Mathematics: An International Journal*, vol. 8, pp. 31–45, 2013, dostęp: 03.02.2021.

[6] D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning," in. The University of Alabama, 1989, ch. 1, pp. 1–2, dostęp: 04.02.2021.

16 Link do repozytorium projektu

Repozytorium na GitHub.