Dokumentacja Projektu
Temat: Zastosowanie algorytmu optymalizacji lokalnej opt-2 dla problemu komiwojażera
Autor: Iwona Fąfara

Sprawozdanie z wykonania projektu

1. Wstęp

Ogólnie znany jest problem komiwojażera (TSP), w którym handlarz mając do odwiedzenia x miast, musi wjechać do każdego z nich dokładnie raz. Jednocześnie stara się on zoptymalizować swoją trasę tak, aby sumarycznie pokonać jak najmniejszą odległość lub zrobić to jak najniższym kosztem. Problem ten należy do klasy problemów NP-zupełnych, co powoduje że dokładny algorytm rozwiązujący ten problem ma złożoność wykładniczą, a co się z tym wiąże, jest nieużyteczny w normalnych zastosowaniach. Z tego powodu, do rozwiązywania problemów tego typu powstało wiele algorytmów przybliżających nas do optymalnego rozwiązania, w których akceptujemy ich niedokładność na rzecz skrócenia czasu wykonywania obliczeń.

2. Model problemu

Podstawą programu jest opieranie się o cykl Hamiltona, czyli cykl w grafie, w którym każdy wierzchołek jest odwiedzony dokładnie jeden raz a ścieżka kończy się w węźle początkowym. Cykl Hamiltona przy każdym wywoływanym grafie jest pierwszą ścieżką kwalifikującą się do optymalizacji.

Optymalizacja ma na celu zminimalizować koszt przejścia całej ścieżki. Jest to możliwe dzięki odczytywaniu i porównywaniu wag krawędzi.

$$min(cost_c)$$

gdzie minimalny koszt całkowity to suma wag po wszystkich krawędziach wchodzących w cykl Hamiltona, który optymalizowaliśmy.

$$cost_c = cost_1 + \ldots + cost_n$$

gdzie $cost_1$ to waga pierwszej ścieżki w cyklu. Przechodzenie po kolejnych krawędziach składa się na otrzymanie całkowitego kosztu $cost_c$ drogi.

Przy przechodzeniu po różnych ścieżkach monitorowany jest czas, który nie podlega optymalizacji, ale przyda się przy zaprezentowaniu wyników testów.

3. Algorytm

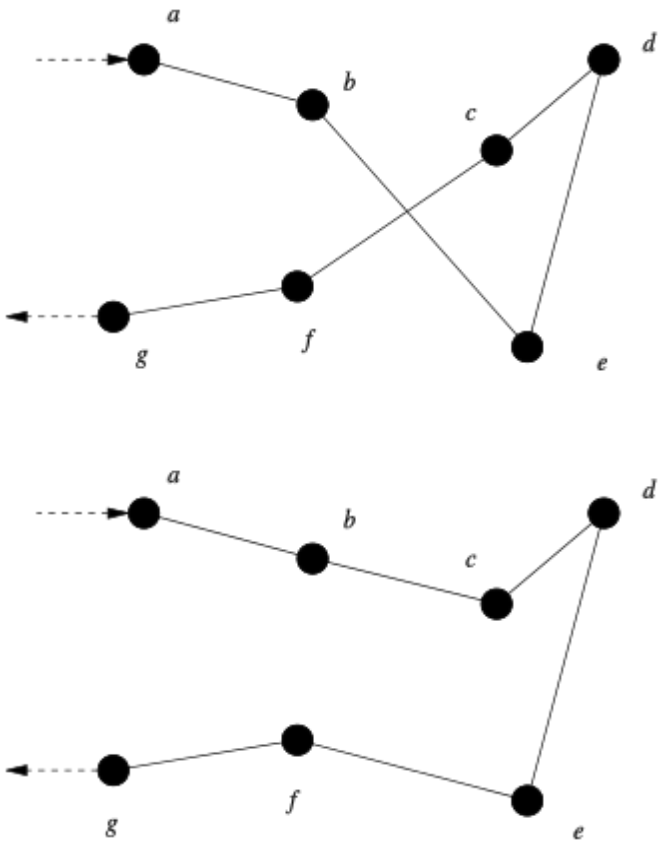
a. rozwiązanie początkowe

Do wygenerowania początkowego rozwiązania, wykorzystano algorytm Nearest Neighbour (algorytm najbliższego sąsiada), dzięki czemu uzyskano cykl Hamiltona. Jest on dobrym wejściem dla naszego algorytmu, ponieważ zapewnia pojawienie się każdego wierzchołka w grafie dokładnie raz na ścieżce. Do dalszej optymalizacji pozostaje nam tylko kolejność wierzchołków, a co się z tym wiąże, wybór ścieżek tak, aby uzyskać sumarycznie jak najmniejszy koszt przejścia.

Wierzchołek startowy sztywno zaczynamy od node = 1 co oznacza, że dla losowego początkowego wierzchołka można by osiągnąć inny graf lub bardziej optymalną ścieżkę dla niego. Jest to funkcjonalność, którą można w przyszłości zaimplementować w celu poprawienia algorytmu. Jednakże w tym projekcie należy pamiętać, że optymalizujemy jedynie cykl dla node = 1 lub innego, który został sztywno wpisany do zmiennej.

b. Algorytm 2-opt

Prosty algorytm optymalizacji lokalnej dla problemu komiwojażera, który szuka przecinające się ścieżki i przepina odpowiadające im wierzchołki tak, aby te ścieżki więcej się nie przecinały. Dobrze obrazuje to poniższa grafika:



(źródło: wikipedia)

Algorytm ten można opisać poprzez pseudokod:

1) **dla każdej pary wierzchołków i, j** w cyklu Hamiltona C wykonaj:
    a) weź wycinek ścieżki C[i, j]
    b) odwróć kolejność wierzchołków na liście
    c) umieść odwróconą listę ponownie w cyklu
    d) sprawdź czy nowy cykl jest krótszy niż poprzedni
        i) jeśli tak: zapisz nowy cykl
        ii) w przeciwnym przypadku: powróć do starego cyklu

Jak widać, powyższy algorytm jest bardzo prosty w budowie, jednak pomimo tego faktu, pozwala na sporą optymalizację pierwotnego cyklu. Złożoność obliczeniowa tego algorytmu wynosi $O(n^2)$, gdzie n jest równe rzędowi rozważanego grafu. Taka złożoność występuje ze względu na generowanie każdej pary dwóch wierzchołków i ich późniejsze sprawdzanie.

c. Wprowadzenie pogorszenia rozwiązania

Wspomniany wcześniej algorytm 2-opt rozszerzono o uwzględnianie k-najlepszych rozwiązań, zamiast tylko jednego. Zabieg ten pozwolił dodatkowo zoptymalizować uzyskiwane ścieżki, ponieważ niektóre optymalizacje, mimo że lokalnie optymalne, mogły blokować inne, które były bardziej korzystne w kolejnych iteracjach.
Niestety, wielkość tego parametru w sposób widoczny wpływała na uzyskiwane czasy, ponieważ dodatkowo zwiększała złożoność algorytmu. Z tego powodu zdecydowano się na ograniczenie ilości ścieżek rozpatrywanych w kolejnych iteracjach do wielkości parametru ‘k’. Czas pracy algorytmu z k-krotnym pogorszeniem jest około k-krotnie dłuższy niż czas potrzebny dla algorytmu bez pogorszenia.

d. Testowane grafy - specyfikacja danych

Do rysowania i optymalizowania ścieżki w grafach potrzebne jest wygenerowanie danych losowych lub zaimportowanie danych węzłów grafu oraz wag między krawędziami.
W projekcie w pliku import_data.py zaimporotowano 7 plików zawierających współrzędne wierzchołków grafów zapisane w różny sposób. Współrzędne wierzchołków reprezentują miasta, w większości położone na terytorium Niemiec. Ze względu na to, że są to rzeczywiste drogi między miastami grafy nie należą do grafów pełnych i ścieżki nie są regularne. Pliki z danymi pochodzą ze strony http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html. Są one w rozszerzeniu .tsp dzięki czemu w projekcie zaimportowano pythonową bibliotekę TSPLIB95. Biblioteka pozwoliła na wyciągnięcie informacji o rzędzie grafu, wagach krawędzi oraz długości krawędzi. Niestety dane źródłowe nie były podane w zunifikowany sposób. Część plików zawierała same wagi krawędzi, macierz sąsiedztwa lub współrzędne geograficzne wierzchołków, z których została obliczona droga między węzłami. Ze względu na parę wariantów importowania danych, w programie zostały zaimplementowane funkcje dla parametrów GEO, czyli wartości euklidesowych oraz dla parametrów zawierających wagi krawędzi.

W projekcie komiwojażera optymalizacja będzie odbywać się na grafach o liczbie wierzchołków: 16, 24, 29, 48, 100 oraz 280.
Grafy są plotowane z odpowiadającym tytułem wykresu dla algorytmu i aktualnej iteracji. Powyżej 50 węzłów, wierzchołki nie są zaznaczone na wykresach.

4. Implementacja

Cały kod stworzony podczas tego projektu można znaleźć w repozytorium na githubie: https://github.com/iwonafafara/BO_TSP_project.

a. **Reprezentacja grafów**
   Do reprezentacji grafów powstały 2 klasy:
   - **Graph** - klasa zapewniająca podstawowe operacje na grafach, przechowuje ilość wierzchołków, tablicę odwiedzin oraz listę sąsiedztwa grafu, która przechowuje jednocześnie wagi ścieżek. Klasa ta zawiera ponadto metody do wyliczenia kosztu przejścia podaną ścieżką, czyszczenia listy odwiedzin oraz obsługi listy sąsiedztwa.
   - **EuclideanGraph** - klasa dziedzicząca z klasy Graph, dodaje możliwość przechowywania współrzędnych euklidesowych wierzchołków w grafie oraz posiada metodę umożliwiającą wygenerowanie ścieżek między tymi wierzchołkami, z wypełnieniem wag poprzez odległości euklidesowe.

b. **Nearest Neighbour**
   Algorytm służący do wygenerowania cyklu Hamiltona w podanym grafie. Zaczynając od pierwszego wierzchołka, wybieramy ścieżkę o najmniejszej wadze prowadzącą do nieodwiedzonego jeszcze wierzchołka i dopisujemy go do listy. Operację powtarzamy do momentu, aż lista nie osiągnie rozmiaru równego ilości wierzchołków, a wtedy sprawdzamy możliwość dojścia do wierzchołka początkowego i dopisujemy go do listy.

c. **2-opt**
   Opis działania algorytmu 2-opt został przedstawiony wcześniej w punkcie 2.b. Z tego powodu skupię się teraz na jego implementacji: funkcja *optimization_2_opt* bierze jako parametry graf, początkowy cykl Hamiltona oraz ilość iteracji bez poprawy kosztu, po których nastąpi przerwanie wykonania algorytmu. Pełni ona funkcję kontrolującą konkretne wykonanie algorytmu 2-opt, który jest realizowany przez funkcję *optimization_2_opt_worker*. Pomocniczo zostały zaimplementowane dodatkowe funkcje: *get_indexes_for_2_opt* i *get_2_opt_reversed_path*, które odpowiednio zwracają pary indeksów dla podlisty do odwrócenia kolejności odwiedzanych wierzchołków i faktycznie dokonują tej podmiany.

d. **2-opt z pogorszeniem**
   Analogicznie do algorytmu 2-opt bez pogorszenia, zostały zaimplementowane funkcje *optimization_2_opt_with_k_deterioration* i *optimization_2_opt_with_k_deterioration_worker*, jednak rozszerzono je o możliwość zwracania k-najlepszych wyników, które są później scalane w

jedną posortowaną listę, która jest obcinana do k elementów przed kolejną iteracją. Powtarzające się listy nie są brane pod uwagę w kolejnej iteracji.

5. Testy i wnioski

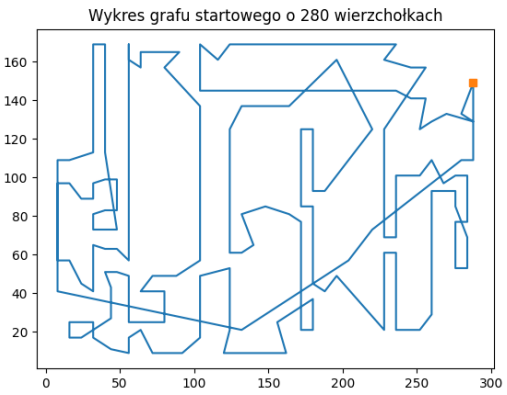
a. Porównanie wyników uzyskanych dla grafów o różnej wielkości dla obu algorytmów

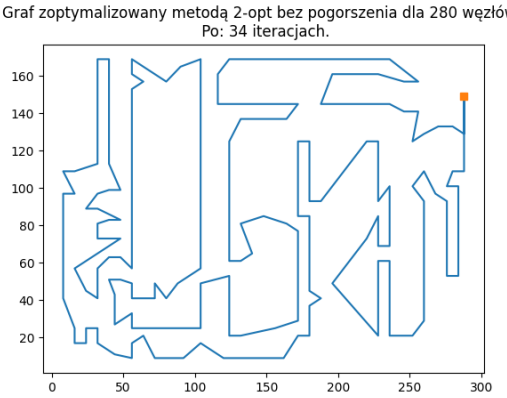
Badania zostały przeprowadzone dla grafów o liczbie wierzchołków do 280. Rozpatrywane wyniki odnoszą się do liczby przebytych iteracji, sumarycznego czasu potrzebnego na obliczenia oraz sumarycznego kosztu przebytego dystansu (wagi krawędzi).
Optymalny koszt został zaciągnięty ze strony opisanej w punkcie 3.d.

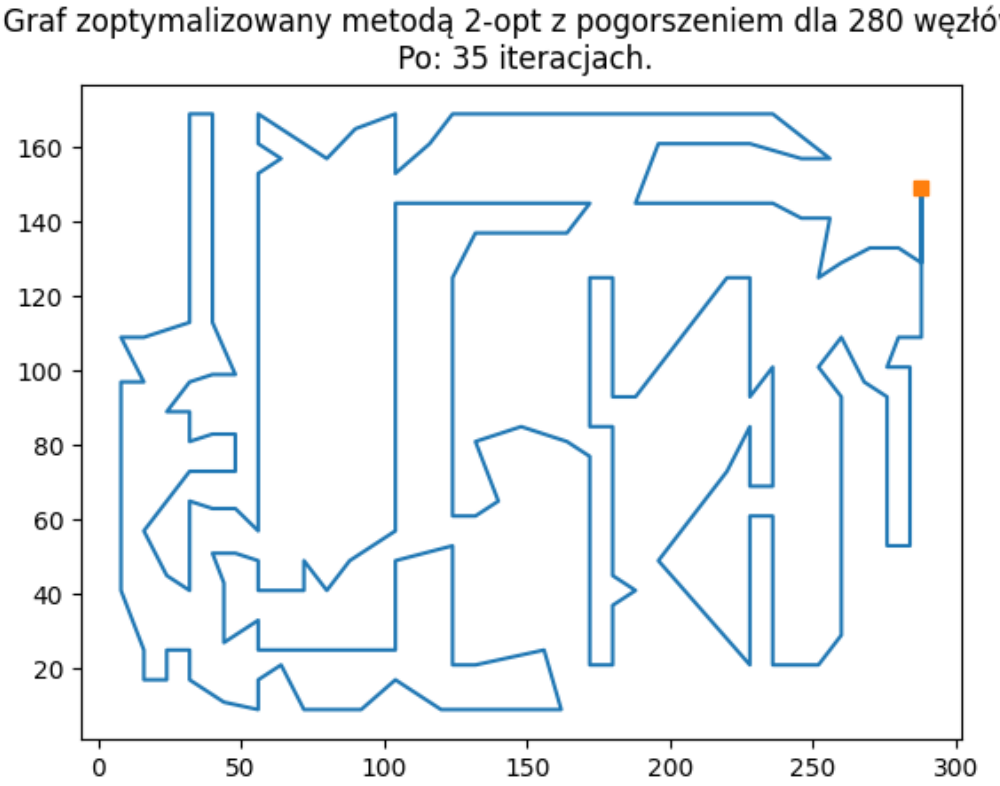
Ilość wierzchołków	16	24	29	48	100	280
Optymalny koszt	82.819	1272	1610	5046	21282	2579
Koszt opt-2	83.605	1426	1659	5403	22775.517	2751.253
Koszt opt-2 z pogorszeniem	80.680	1379	1646	5197	22803.584	2732.659
Liczba iteracji	5	5	4	10	19	34
Liczba iteracji z pogorszeniem	8	6	6	12	17	35
Czas [ms]	3.992	26.006	34.052	618.688	21904.519	2418613.782
Czas z pogorszeniem [ms]	45.501	123.030	271.581	4268.508	99839.990	13498723.349

Dla grafu num.of.nodes = 16 i algorytmu z pogorszeniem różnica w wyliczonym koszcie i koszcie optymalnym może wynikać ze specyficznych danych opisujących wierzchołki grafu. W tym przypadku były to współrzędne euklidesowe. Ponieważ nie były podane wagi ścieżek, to zostały one wygenerowane pomiędzy dwoma każdymi wierzchołkami. Z niewiadomych przyczyn wynik optymalny był podany jako kwadrat odległości. Myślę, że ze sposobu generowania ścieżek i ich wag, może wynikać rozbieżność pomiędzy otrzymanym wynikiem a wynikiem oficjalnym.

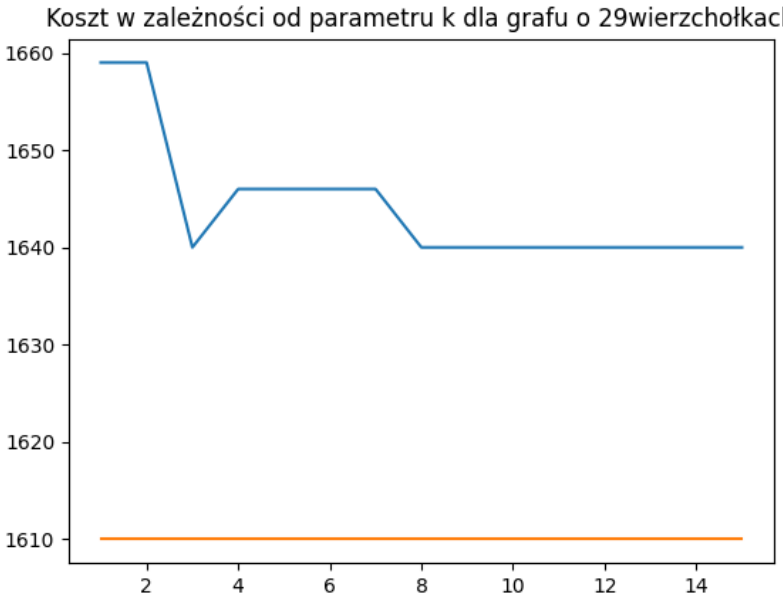
Przedstawione poniżej grafiki obrazują graf num_of_nodes = 280 z cyklem Hamiltona wygenerowanym przez algorytm Nearest Neighbour, graf otrzymany po optymalizacjach bez k-pogorszenia oraz graf otrzymany w wyniku algorytmu z pogorszeniem dla parametru ‘k’.





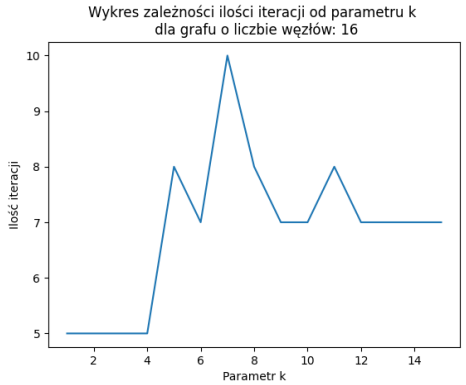


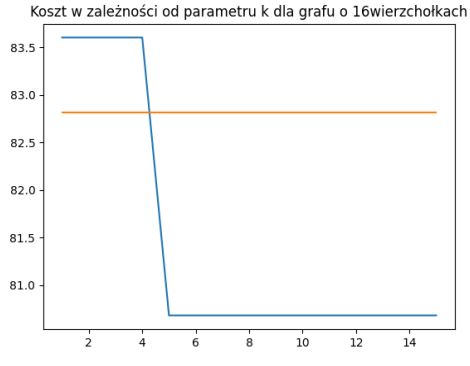
b. Uzyskany koszt w zależności od wartości parametru ‘k’ dla grafu o 29 wierzchołkach:

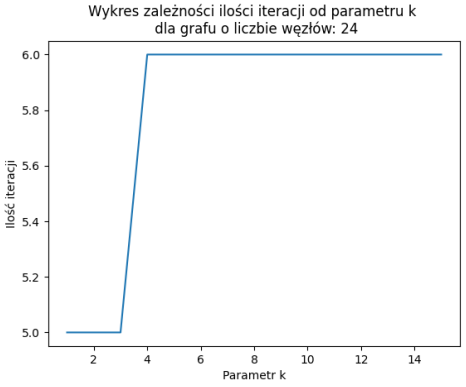


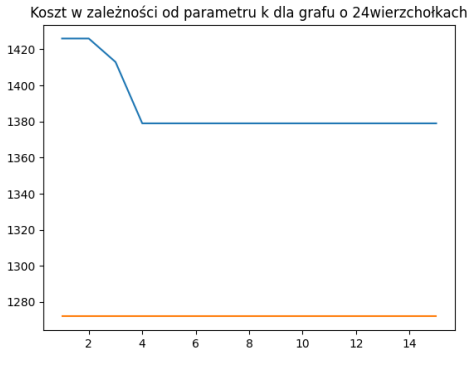
Możemy zaobserwować, że od pewnej wartości parametru ‘k’, uzyskiwany wynik się już nie poprawia. Zaskoczenie budzi fakt, że dla k z przedziału <4, 7> uzyskiwany wynik jest gorszy niż dla k=3, jednak dla większych wartości, wynik jest już stabilny.

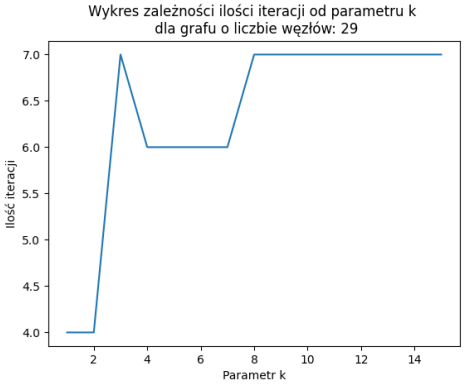
c. **stosunek ilości potrzebnych iteracji do parametru “k” dla różnych wielkości grafu. Zestawiono go z kosztem, jaki został obliczony przez algorytm.**

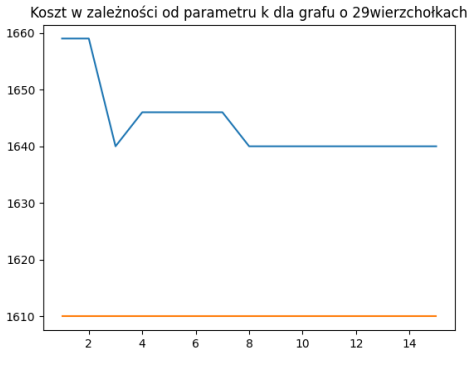


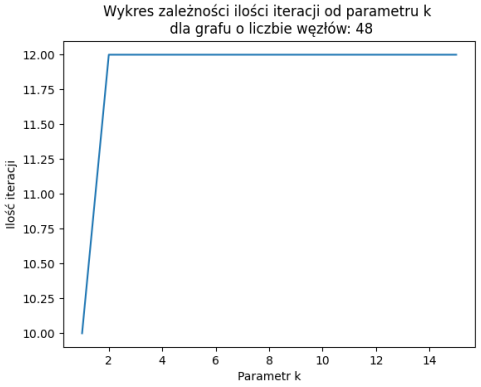


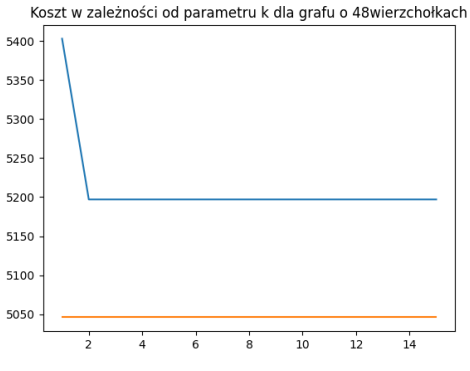


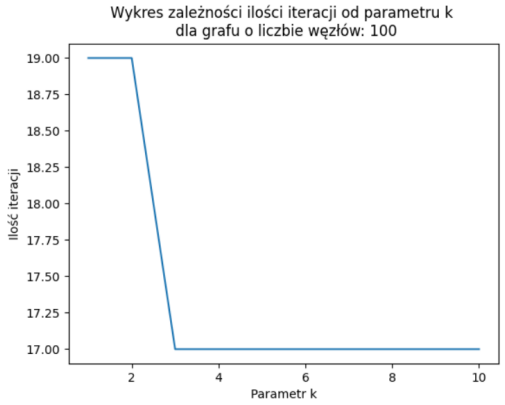


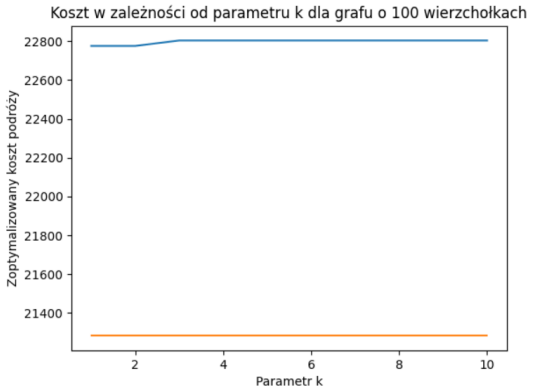












Możemy zaobserwować, że dla małych grafów, występuje **korelacja ilości wykonanych iteracji z jakością uzyskanego rozwiązania**. Widzimy, że graf o 16 wierzchołkach zachowuje się w bardzo nieoczywisty sposób, ponieważ w żaden sposób nie da się powiązać wzrostu/spadku ilości iteracji ze wzrostem parametru k. Co ciekawe, w dużym grafie (100 wierzchołków), możemy zaobserwować spadek ilości potrzebnych iteracji z 19 do 17 dla k=3 i od tego momentu ilość iteracji pozostaje stabilna. Jednocześnie możemy zauważyć, że wraz ze spadkiem ilości iteracji, nieznacznie rośnie koszt uzyskanego rozwiązania.

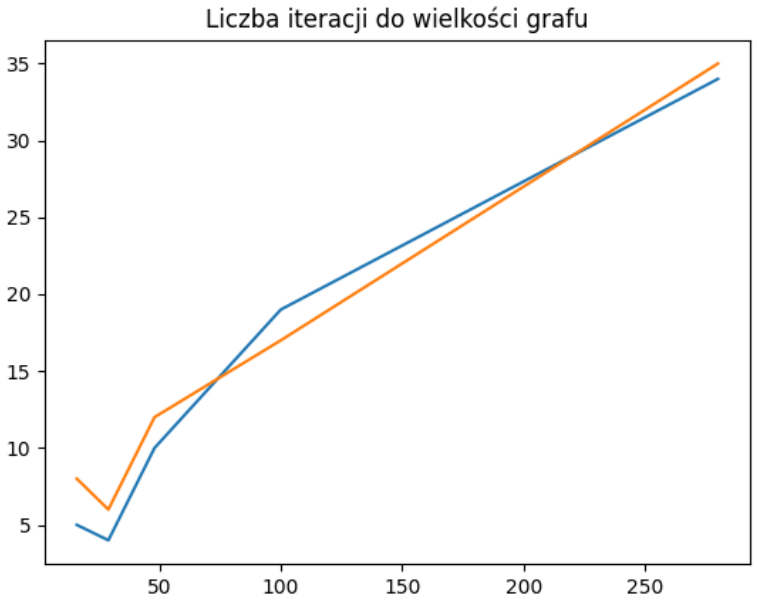
Warto zauważyć, że k=1 jest szczególnym przypadkiem, ponieważ odpowiada realizacji algorytmu 2-opt bez pogorszenia.

d. Rozbieżność od optymalnego rozwiązania

|V|	Optimum	opt-2	opt-2 k=5	Błąd opt-2	Błąd opt-2 k=5
16	82.819	83.605	80.68	0.95%	-2.58%
24	1272	1426	1379	12.11%	8.41%
29	1610	1659	1646	3.04%	2.24%
48	5046	5403	5197	7.07%	2.99%
100	21282	22775.517	22803.584	7.02%	7.15%
280	2579	2751.253	2732.659	6.68%	5.96%

Widzimy, że w uzyskanym wyniku maksymalny błąd względny wynosi około 12%, jednak po wykonaniu algorytmu z pogorszeniem k=5, maksymalny błąd wynosi już tylko niecałe 8.5%. Jedynie dla grafu o 100 wierzchołkach został zanotowany nieznaczny wzrost kosztu po zastosowaniu parametru “k”, jednak przypadek ten został już omówiony w poprzednim teście, razem z ilością iteracji dla tego algorytmu. Wyniki te są zadziwiająco dobre jak na prostotę tego algorytmu. W dwóch przypadkach poprzez zastosowanie parametru ‘k’ udało się uzyskać aż 4-procentową poprawę wyniku.

e. Porównanie ilości potrzebnych iteracji do wielkości grafu

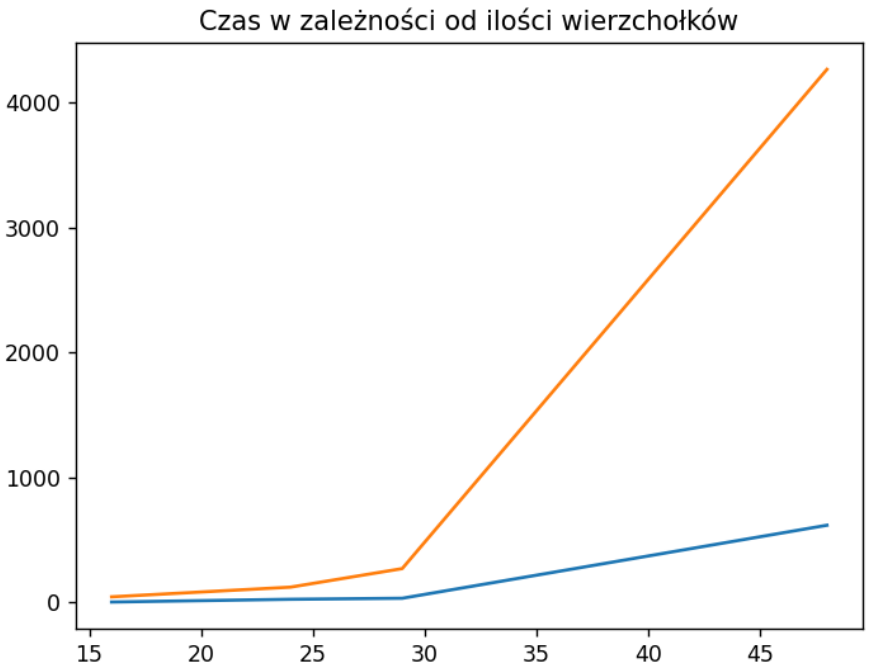


Na wykresie widzimy ilość potrzebnych iteracji do znalezienia rozwiązania w zależności od wielkości grafu. Niebieską linią zaznaczono iteracje dla algorytmu opt-2, natomiast pomarańczową pokazano iteracje dla algorytmu opt-2 z pogorszeniem. Widzimy, że tylko dla grafu o 100 wierzchołkach, parametr k zmniejszył ilość iteracji, jednak (jak pokazano wcześniej) pogorszył zarazem końcowy wynik.

f. Testy dla grafu o 1060 wierzchołkach

W projekcie został zaimportowany graf o liczbie wierzchołków 1060, jednak czas wykonania algorytmu dla niego, okazał się być zbyt duży, aby możliwe było jego przetestowanie. Z tego powodu nie został uwzględniony w testach.

g. Porównanie sumarycznych czasów wykonywanych się algorytmów opt-2 i opt-2 z k=5 pogorszeniem dla różnej wielkości grafu.



Niebieska linia → wykres algorytmu opt-2
Pomarańczowa linia → wykres algorytmu opt-2 z pogorszeniem
Jest to zależność kwadratowa, jednak nie widać tego wprost ze względu na małą ilość grafów (16, 24, 29, 48).

6. Podsumowanie

Otrzymane wyniki pokazały, że prosty algorytm optymalizacji lokalnej jest w stanie poradzić sobie z tak złożonym problemem jak problem komiwojażera. W znacznie krótszym czasie otrzymujemy wynik, który nie odbiega znacząco od optymalnego rozwiązania (różnica błędu mniejsza niż 13%). Rozpatrując nie tylko lokalnie optymalną ścieżkę, ale też kilka gorszych (w zależności od wielkości parametru k) jesteśmy w stanie jeszcze bardziej usprawnić ten algorytm, dzięki czemu różnica od optymalnego rozwiązania jest mniejsza niż 9%. Niestety duży wpływ na zaimplementowany algorytm ma sposób generowania początkowego cyklu Hamiltona. Szczególnie wybór pierwszego wierzchołka w algorytmie Nearest Neighbour. Istnieje prawdopodobieństwo, że początkowe wybranie cyklu algorytmem losowym dawałoby różne wyniki końcowe, potencjalnie lepsze.