

**AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU**

WYDZIAŁ NAUK INŻYNIERYJNYCH

PRACA DYPLOMOWA

**ANALIZA I OCENA EFEKTYWNOŚCI ALGORYTMÓW
ROZWIĄZANIA PROBLEMU KOMIWOJAŻERA**

Autor: Filip Rzepiela

Kierunek: Informatyka

Nr albumu: 29757

Promotor: dr inż. Józef Zieliński

**Akceptacja promotora:
data i podpis**

NOWY SĄCZ 2024

Spis treści

Wstęp	5
1. Problem komiwojażera	6
1.1. Geneza problemu komiwojażera	6
1.2. Opis problemu	21
1.3. Rodzaje problemów	25
1.4. Podstawowe pojęcia	26
1.4.1. Graf	26
1.4.2. Cykl Eurela	27
1.4.3. Twierdzenie Eurela	28
1.4.4. Graf eurelowski i półeurelowski	28
1.4.5. Cykl Hamiltona	29
1.4.6. Graf Hamiltonowski	30
1.4.7. Problem cyklu Hamiltona	33
2. Cel i zakres pracy	35
3. Metodyka badań	36
4. Przegląd algorytmów	38
4.1. Algorytmy k-optymalne	38
4.1.1. Algorytm dwu-optymalny	39
4.1.2. Algorytm trój-optymalny	40
4.2. Algorytm zachłanny	41
4.2.1. Algorytm Kruskala	43
4.2.2. Algorytm Prima	46
4.3. Algorytm selekcji krawędzi	47
4.4. Algorytm genetyczny	50
4.5. Algorytm mrówkowy	55
4.6. Algorytm Christofidesa	59
4.7. Algorytm Little'a	61
5. Przedstawienie środowiska pracy	67
5.1. Visual Studio Community 2022	67
5.2. Język C++	68
5.3. Jupyter Notebook	68
5.4. Język Python	69
5.5. MatLab	69
5.6. System kontroli wersji GitHub	70

6.	Implementacja komputerowa.....	72
6.1.	Generator grafów losowych	72
6.2.	Implementacja algorytmu selekcji krawędzi.....	75
6.3.	Implementacja algorytmu dwu-optymalnego.	86
6.4.	Implementacja algorytmu trój-optymalnego.....	96
6.5.	Implementacja algorytmu Christofidesa	109
7.	Badanie porównawcze algorytmów	125
8.	Wyniki badań	135
9.	Podsumowanie i wnioski	157
	Bibliografia	158
	Spis rysunków	162
	Spis tabel	167
	Spis załączników.....	168

Wstęp

Obecnie szybkość przetwarzania informacji oraz zdolność do efektywnej optymalizacji procesów mają decydujące znaczenie dla konkurencyjności i wydajności przedsiębiorstw oraz organizacji na całym świecie. W erze cyfryzacji, astronomiczne ilości danych są generowane, przetwarzane i analizowane niemalże w czasie rzeczywistym. Algorytmy optymalizacyjne stanowią fundament nie tylko w informatyce, ale również umożliwiają zarządzanie zasobami, planowanie logistyki, produkcji i innych aspektów działalności gospodarczych. Informatyka zajmując się optymalizacją kombinatoryczną, poświęca szczególną uwagę problemowi komiwojażera (TSP – Travelling Salesman Problem). Problem ten, mimo pozornej prostoty polegającej na odnalezieniu najkrótszej możliwej ścieżki przechodzącej przez określoną liczbę punktów i powracającej do punktu wyjścia, jest jednym z najbardziej złożonych i trudnych wyzwań obliczeniowych, uznawanym za NP-trudny. Oznacza to, że przy większej liczbie punktów nie istnieje algorytm, który potrafi rozwiązać problem w czasie wielomianowym¹ dla wszystkich możliwych instancji.

W pierwszym rozdziale przedstawiony został problem komiwojażera. Rozpoczęto od genezy problemu na przestrzeni lat, począwszy od czasów prehistorycznych, aż do dziś. Omówiony został opis omawianego zagadnienia, wyjaśnione zostały rodzaje występujących problemów rozpoczynając od klasy najniższej do najwyższej NP-trudnej. W ramach zgłębienia tematu przedstawione zostały również podstawowe pojęcia, za pomocą których zrozumienie idei problemu komiwojażera było prostsze do przyswojenia. Rozdział drugi skupia swoją uwagę na przedstawieniu celów, jakie w tej pracy miały zostać osiągnięte. Kolejny rozdział zawiera w sobie metodykę przeprowadzenia badań, która oparta została na wszelkiej dostępnej literaturze, bez ograniczania się do pozycji wyłącznie polskojęzycznych. Rozdział czwarty to przedstawienie znanych algorytmów, za pomocą których można podjąć próbę rozwiązania NP-trudnego problemu komiwojażera. W rozdziale piątym dokonano przedstawienia języków programowania oraz środowisk pracy, za pomocą których wykonane zostały wybrane programy. Rozdział szósty z kolei przedstawia implementację wyselekcjonowanych algorytmów. Rozdział siódmy przedstawia proces przeprowadzania badań na wykonanych programach, w różnych językach oraz na różnych sprzętach. W rozdziale ósmym omówione zostały wyniki przeprowadzonych badań. Pracę zwieńczają podsumowanie i wnioski.

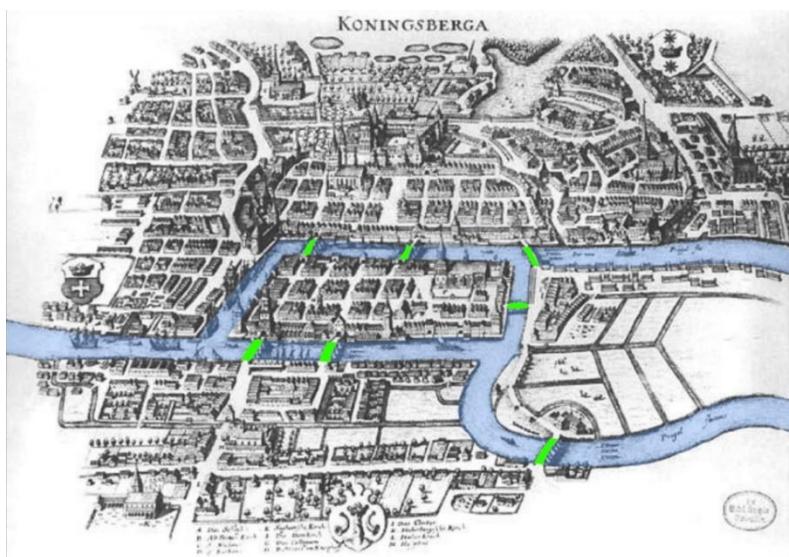
¹ Czas wielomianowy – klasa złożoności algorytmów, których czas działania rośnie w tempie będącym funkcją wielomianową w stosunku do rozmiaru danych wejściowych.

1. Problem komiwojażera

Rozdział pierwszy skupia się na przedstawieniu historii problemu komiwojażera od czasów prehistorycznych do współczesnych. W drugiej części rozdziału objaśniono problem komiwojażera. Trzeci podrozdział przedstawia rodzaje problemów, zaczynając od prostych, rozwiązywalnych, a kończąc na nierozwiązywalnych. W czwartej części przedstawiono podstawowe pojęcia związane z omawianym problemem, aby lepiej zrozumieć jego istotę.

1.1. Geneza problemu komiwojażera

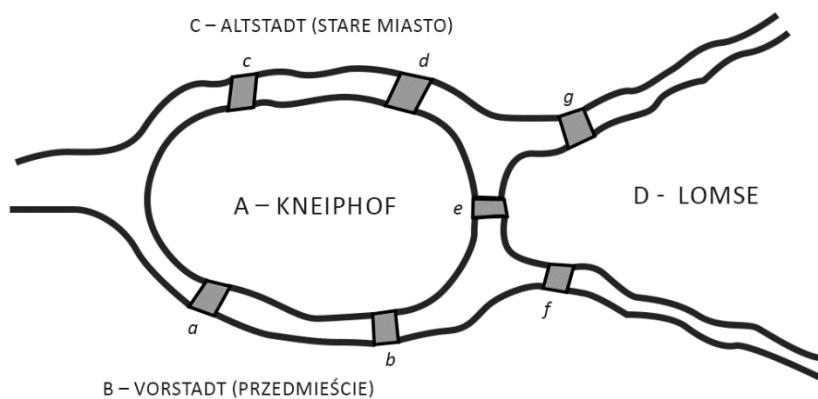
(Cook, 2012) Konieczność optymalizacji tras sięga czasów prehistorycznych, kiedy to jaskiniowcy rozwiązywali małe wersje problemu komiwojażera w czasie polowań oraz zbieractwa, niemniej jednak nie został on sformułowany w sposób matematyczny. Na przestrzeni wieków przedstawiciele różnorakich zawodów starali się zaplanować trasę podróży. (Wikipedia, 2023) W 1741 roku Leonhard Euler opublikował pracę „Solutio problematis ad geometriam situs pertinentis w Commentarii academieae scientiarum Petropolitanae”², która opisywała problem objazdowy. (Cook, 2012) Euler we wspomnianym wcześniej dziele opisał problem mostów królewieckich. Rysunek 1 przedstawia plan miasta Królewiec, w którym Euler przeprowadził badanie. Kolorem niebieskim zaznaczona została rzeka Pregel, obecnie nazywana Pregoła, natomiast kolorem zielonym zaznaczone zostały mosty.



Rysunek 1. Plan miasta Królewiec z czasów Eulera.
[źródło: (Kohlstedt, 2022)]

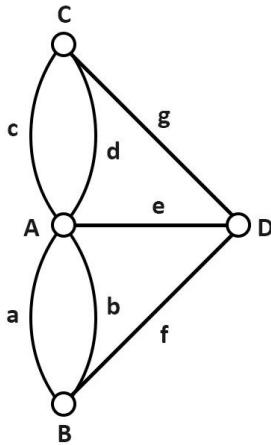
² „Solutio problematis ad geometriam situs pertinentis w Commentarii academieae scientiarum Petropolitanae.” – z łacińskiego „Rozwiązywanie problemu dotyczącego geometrii stanowiska w Komentarzach Petropolitan Academy of Sciences.”

W epoce Eulera rzekę przecinało siedem mostów łączących kolejno wyspę Kneiphof ze Starym Miastem (region na północy), Przedmieściem (region południowy) oraz Lomse (wschodnia wyspa), a także wschodnią wyspę z Przedmieściem oraz Starym Miastem. Wyzwaniem, przed jakim stawali obywatele miasta, było przekroczenie podczas spaceru każdego z mostów wyłącznie raz. Analizy i próby rozwiązania problemu podjął się Euler, początkowo szkicując plan miasta, rzekę oraz mosty, a następnie opisał części Królewca jako litery A, B, C oraz D, a także mosty Green, Köttel, Krämer, Schmiede, Honey, Lomse i Wood jako litery a-g, co zostało przedstawione na rysunku 2.



Rysunek 2. Przedstawienie mostów w Królewcu według Eulera.
[źródło: opracowanie własne na podstawie (Cook, 2012)]

Zaproponowane przez autora etykiety były wystarczające do opisania dowolnej trasy np. z punktu A do C przez most c. Warto zauważyć, że przedstawione przez Leonharda Eulera argumenty były w pełni oparte na manipulacji trasami jak ciągami symboli, natomiast wielkość lądowa nie odgrywała żadnej roli, co pozwoliło przedstawić mosty królewieckie na postawie prostego wykresu, w którym części miasta stanowią punkty A-D, natomiast mosty A-G przedstawione zostały jako linie, za pomocą których punkty zostały połączone, co zilustrowane zostało na rysunku 3.



Rysunek 3. Wykres przedstawiający reprezentację Mostów Królewieckich.

[źródło: opracowanie własne na podstawie (Cook, 2012)]

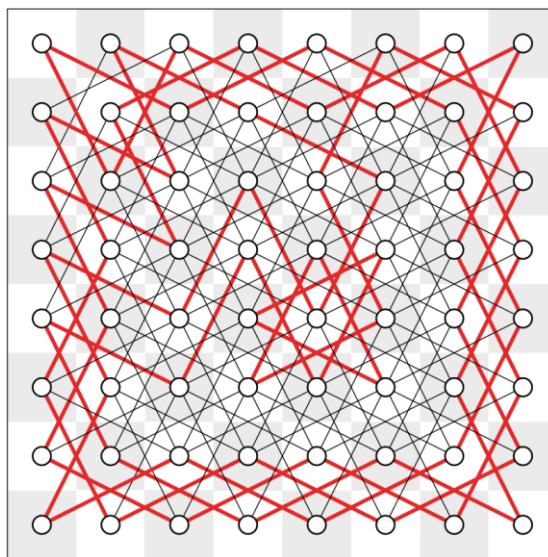
Według przedstawionego powyżej wykresu można zauważyć, że spacer po Królewcu opierał się na przemieszczaniu z wierzchołka do wierzchołka za pomocą krawędzi przebiegającej między nimi. Przykładowo dla zaplanowanego spaceru z punktu B do D, przez każdy z siedmiu mostów, trasa wygląda w następujący sposób: B a AcCg De Ab B f D. W przykładowej trasie spacerowej w punkcie B spotkały się trzy krawędzie – a, b oraz f, w punkcie A natomiast dochodziło do spotkania czterech krawędzi a, c, e oraz b. Analizując kolejne punkty, w wierzchołku C spotykało się krawędź c oraz g, natomiast w D ponownie dochodzi do zetknięcia się ze sobą trzech krawędzi, mianowicie g, e oraz f. Euler zaobserwował, że liczby spotkań w danym punkcie mają wzór nieparzysty-parzysty-parzysty-nieparzysty. Podczas spaceru pomiędzy dwoma różnymi punktami, spotykana była nieparzysta liczba krawędzi, natomiast z innymi wierzchołkami stykają się parzystą ilością krawędzi. Jeżeli pod uwagę wzięty zostanie spacer zaczynający się i kończący w tym samym punkcie, to w tej sytuacji każdy z punktów spotykał parzystą ilość krawędzi. Wspomniana obserwacja pomogła zakończyć debatę mieszkańców Królewca, ze względu na fakt, że wszystkie cztery części miasta spotykają się z nieparzystą ilością krawędzi i nie istnieje możliwość przeprowadzenia spaceru z wykorzystaniem każdej z krawędzi wyłącznie raz. Teoria Eulera stanowiła podwaliny pod teorię grafów, która stała się kluczowa dla rozwoju problemu komiwojażera.

W późniejszym czasie Euler opisał kolejny problem objazdowy – problem wędrówki rycerza w szachach. Celem rzeczonego wyzwania było znalezienie sekwencji ruchów rycerza na szachownicy, która umożliwi odwiedzenie co drugiego pola dokładnie raz, zaczynając i kończąc na polu startowym. Zaproponowane przez autora rozwiązanie, w którym kolejność każdego ruchu została opisana liczbowo w polach szachownicy, przedstawiono na rysunku 4.

42	57	44	9	40	21	46	7
55	10	41	58	45	8	39	20
12	43	56	61	22	59	6	47
63	54	11	30	25	28	19	38
32	13	62	27	60	23	48	5
53	64	31	24	29	26	37	18
14	33	2	51	16	35	4	49
1	52	15	34	3	50	17	36

Rysunek 4. Rozwiązywanie Eurela problemu wędrówki rycerza
[źródło: opracowanie własne na podstawie (Cook, 2012)]

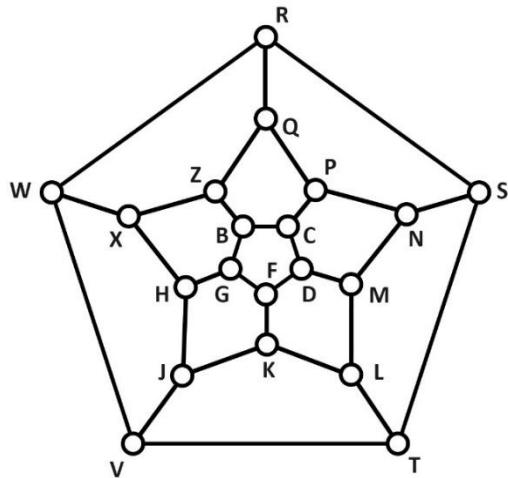
W przypadku problemu wędrówki rycerza dla każdego z pól szachownicy ustalony został wierzchołek z dwoma wierzchołkami połączonymi krawędzią, w przypadku gdy rycerz chciał przemieścić się między polami za pomocą tylko jednego ruchu. Podobnie jak w przypadku mostów królewieckich stanowi to spacer zamknięty, ponieważ należy rozpocząć i zakończyć w tym samym punkcie, przechodząc przez każdą krawędź tylko raz, co zostało przedstawione na rysunku 5.



Rysunek 5. Trasa rycerska na wykresie szachownicy.
[źródło: (Cook, 2012)]

Wiek po Eulerze temat wycieczek po konkretnym wykresie zainteresował kolejnego uczonego, Sir Williama Rowana Hamiltona, który rozważał sposoby odwiedzenia wszystkich

dwudziestu punktów narożnych dwunastościanu dwunastościennej bryły platońskiej. Podczas analizy problemu opracował abstrakcyjny rysunek, znany jako Icosian, w którym linie reprezentują geometryczne krawędzie dwunastościanu, natomiast okręgi przedstawiają jego narożniki. Graf opracowany przez Hamiltona przedstawiony został na rysunku 6.



Rysunek 6. Icosian.
[źródło: opracowanie własne na podstawie (Cook, 2012)]

W grafie Hamiltona wycieczki rozpoczynały się od wierzchołka do wierzchołka, podróżując wzduż krawędzi wykresu. W Icosianie Hamilton wprowadził system algebraiczny do reprezentowania możliwych ścieżek w grafie. Przyjął symbole i , κ oraz λ , których zadaniem było spełnianie czterech następujących równań:

$$i^2 = 1 \quad (1)$$

$$\kappa^3 = 1 \quad (2)$$

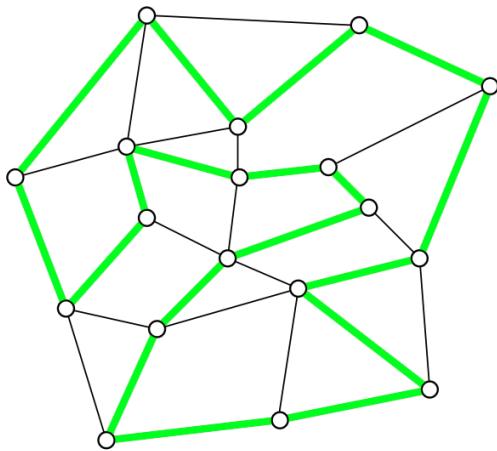
$$\lambda^5 = 1 \quad (3)$$

$$\lambda = i\kappa \quad (4)$$

Równanie (1) symbolizuje operację, która przeprowadzana dwukrotnie nie zmienia położenia na grafie, czyli był to jakby powrót do punktu wyjścia. Równanie (2) natomiast reprezentuje operację, która po trzykrotnym wykonaniu również prowadzi z powrotem do punktu wyjścia, z kolei równanie numer (3) wskazuje na to samo dla operacji wykonanej pięciokrotnie. Ostatnie równanie (4) wskazuje na relację między tymi operacjami.

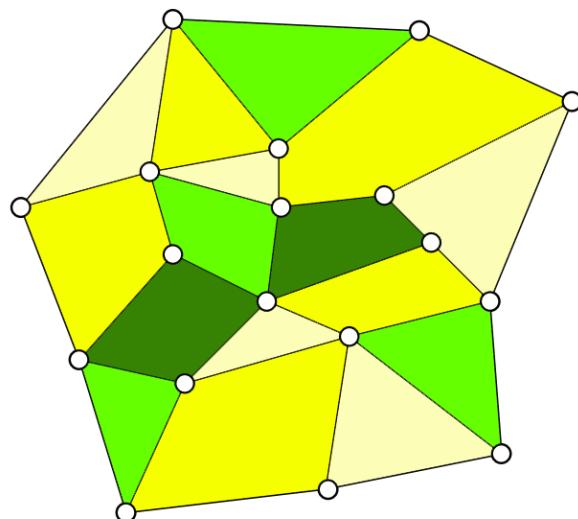
Przeprowadzone przez Hamiltona abstrakcje pozwoliły dokonać manipulacji ścieżkami w Icosianie bez potrzeby wizualizowania każdego etapu, co było przełomowym rozwiązaniem dla jego przyszłych prac nad quaternionami. Na podstawie Icosianu Hamilton opracował grę ikozjańską, która stanowiła nie tylko rozrywkę matematyczną, ale była również platformą badań nad strukturami grafów. Celem gry było odnalezienie ścieżki przechodzącej przez każdy z wierzchołków dwunastoscianu Hamiltona dokładnie raz.

Głównym aspektem wspomnianej gry było zastosowanie idei obwodów, które później znalazły swoje zastosowanie w definicji cykli Hamiltona. (Wilson, 2000) Cykl Hamiltona to zamknięta ścieżka w grafie, w której zbiór wierzchołków, wykluczając punkt startowy oraz końcowy, zostały odwiedzane wyłącznie raz. Cykl rozpoczynał się i kończył w tym samym wierzchołku, odwiedzając pozostałe wyłącznie jeden raz. W przypadku odnalezienia cyklu Hamiltona o minimalnej sumie wag odwiedzonych krawędzi rozwiązany zostawał problem komiwojażera. (Cook, 2012) Określenie czy graf posiada obwód hamiltonowski, czy też nie stanowi problem NP-zupełnym. W 1879 roku Alfred Kempe opublikował dowód twierdzenia o czterech kolorach, który był akceptowany przez matematyków przez prawie dekadę. Strategia Kempa polegała na tym, że każda sytuacja, w której cztery kolory mogłyby być konieczne, mogła być przekształcana za pomocą tych łańcuchów, co pozwalało zachować zgodność kolorów wzdłuż granic regionów. Zainspirowany dowodem Kempa szkocki fizyk i matematyk Peter Guthrie Tait przypuszczał, że pewien typ wykresów ma zawsze cykl hamiltonowski. Aby zauważyć związek pomiędzy podróżowaniem a kolorowaniem mapy należy wyobrazić sobie granice poszczególnych regionów mapy jako krawędzie grafu, natomiast punkty przecięcia jako wierzchołki. Jeżeli przez dany wykres przechodzi cykl Hamiltona, to wykres graniczny umożliwia pokolorowanie mapy. Rysunek 7 przedstawia przykład grafu, przez który przechodzi cykl Hamiltona.



Rysunek 7. Cykl Hamiltona w grafie.
[źródło: opracowanie własne na podstawie (Cook, 2012)]

Kolorem zielonym na powyższym rysunku przedstawione zostały krawędzie tworzące obwód Hamiltona. Można zauważyć, że obwód nie przecina się sam ze sobą, zatem posiada granice zewnętrzne oraz wewnętrzne. Krawędzie graniczne od wewnątrz przecinają obszar wewnętrzny. Zatem istnieje możliwość pokolorowania tych obszarów dwoma kolorami, zmieniając kolor za każdym razem, gdy przekroczona zostanie jedna z krawędzi nieobwodowych. Taką samą zasadę można było zastosować do pokolorowania dwoma kolorami obszarów zewnętrznych znajdujących się poza obwodem Hamiltona. W rezultacie uzyskana została mapa czterokolorowa. W przedstawionym na rysunku 8 przykładzie obszary znajdujące się w cyklu Hamiltona posiadają kolor ciemnożółty oraz jasnożółty, natomiast pozostałe obszary kolor ciemnozielony i jasnozielony.



Rysunek 8. Pokolorowana mapa cyklu Hamiltona.
[źródło: opracowanie własne na podstawie (Cook, 2012)]

Peter Guthrie Tait miał świadomość, że nie wszystkie mapy posiadają w granicach obwody Hamiltona, jednak dostępne sposoby dały możliwość ograniczenia problemu czterech kolorów do map w taki sposób, że każdy wierzchołek wykresu granicznego spotyka dokładnie trzy krawędzie. Należało założyć, że graf brzegowy nie może podzielić wykresu na dwie części poprzez usunięcie wierzchołków. Tait zakładał, że stosując ograniczenie do trzech połączonych map sprawi, że obwody Hamiltona będą zawsze dostępne. Z biegiem czasu okazało się, że zarówno teoria Alfreda Kempe, jak i Petera Guthrie Tait'a okazały się błędne. Problem czterech kolorów jest bardzo złożony, a jego dowód został odnaleziony dopiero w 1976 roku, kiedy to Kenneth Appel i Wolfgang Haken udowodnili twierdzenie o czterech barwach, co było przełomem w długiej historii tego problemu. (Wikipedia, 2024) Dowód Appela i Hakena opierał się na redukcji problemu do mniejszej liczby niezmiennych konfiguracji, które mogą wystąpić na mapie. Zidentyfikowali 1936 niezmiennych konfiguracji, które musiały zostać sprawdzone. W celu dokonania serii skomplikowanych sprawdzeń wykorzystali oni komputer, aby wykazać, że każda z zidentyfikowanych konfiguracji mogła zostać pokolorowana czterema kolorami, tak aby nie naruszyć zasad twierdzenia. Ostateczny dowód twierdzenia wymagał także dowiedzenia, że każda możliwa mapa może być zredukowana do jednej z tych konfiguracji. Ostatecznie przyjęto następującą definicję twierdzenia o czterech barwach, zgodnie z którą w grafie planarnym G składającym się z wierzchołków V i krawędzi E można przypisać każdemu wierzchołkowi jedną z czterech liczb k_1, k_2, k_3, k_4 , odpowiadającym kolorom, tak aby żadne dwa sąsiadujące wierzchołki, czyli wierzchołki połączone bezpośrednio krawędzią, nie miały przypisanego tego samego koloru. Matematycznie relację tę opisują następujące wzory:

$$k: V \rightarrow \{k_1, k_2, k_3, k_4\} \quad (5)$$

$$\forall_{\{v_1, v_2\} \in E} (k(v_1) \neq k(v_2)) \quad (6)$$

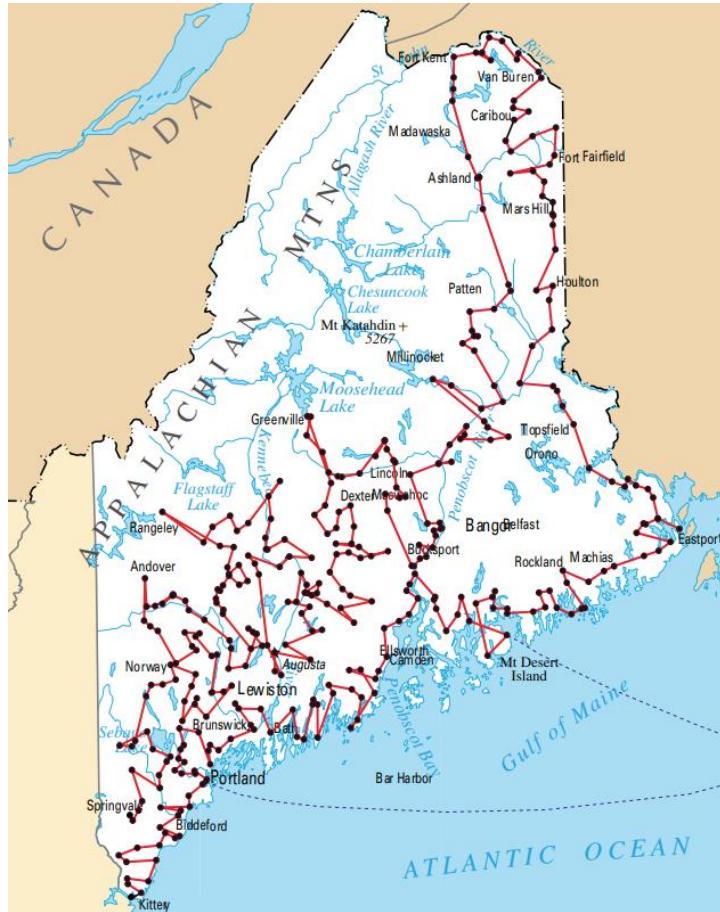
Wzór (5) opisuje funkcję kolorowania k , która przypisuje wierzchołkowi V w grafie jedną z czterech wartości k_1, k_2, k_3, k_4 . Te wartości reprezentują cztery różne kolory. Funkcja ta jest aplikacją, czyli przyporządkowaniem, które dla każdego punktu (wierzchołka) grafu planarnego przypisuje dokładnie jeden z czterech dostępnych kolorów. Wzór (6) natomiast głosi, że dla każdej pary wierzchołków v_1 i v_2 , które są połączone krawędzią E , czyli sąsiadują ze sobą na grafie, kolor przypisany do v_1 jest różny od koloru przypisanego do v_2 . Symbol \forall oznacza "dla wszystkich", co w tym kontekście oznacza, że każda krawędź w grafie musi łączyć wierzchołki różnych kolorów.

(Cook, 2012) Euler wraz z Hamiltonem w swoich badaniach skupili się na badaniu podróży, jednak ich badania były dalekie od problemu sprzedawcy w podróży, który oczekiwany był jak najkrótszej długości. Tematem tym zainteresował się Karl Menger, który w latach dwudziestych XX wieku zajmował się badaniem technik pomiaru długości krzywych w przestrzeni. Badania te stały się inspiracją dla ogłoszenia bliskiego problemowi komiwojażera problemu posłańca. Problem posłańca to zadanie matematyczne i logistyczne, które na co dzień napotyka wielu listonoszy i podróżników. Polega na znalezieniu najkrótszej trasy łączącej określoną liczbę punktów, między którymi z góry wiadomo, jakie są odległości. Celem problemu posłańca było wytyczenie ścieżki, która połączy te punkty w taki sposób, aby cała droga była jak najkrótsza, nie wracając przy tym do punktu wyjścia. Problem ten można łatwo uogólnić do problemu komiwojażera poprzez dodanie fikcyjnego miasta, które służy jako punkt łączący koniec i początek trasy. Przyjmuje się, że koszt dojazdu z tego dodatkowego punktu do każdego z rzeczywistych miast jest zerowy, dzięki czemu nie wpływa on na wybór miejsca startowego ani końcowego na trasie. W ten sposób, mimo że zadanie wydaje się być zmodyfikowaną wersją TSP, jest ono z nim ściśle powiązane i wymaga podobnych technik rozwiązania.

Zagadnienie problemu posłańca, które do tej pory rozważano głównie z punktu widzenia codziennej praktyki i logistyki, zaczęło przechodzić fascynującą transformację za sprawą naukowców, takich jak Merrill Flood. Zastosowane przez Flooda podejście do kwestii optymalizacji, pomimo zakorzenienia w konkretnych potrzebach transportowych — a konkretniej do próby zastosowania matematycznych metod w celu ulepszenia trasowania autobusów szkolnych — szybko zyskało na znaczeniu w teoretycznych badaniach matematycznych. Zarówno Flood, jak i prace Hasslera Whiteneya zaowocowały otwarciem nowego rozdziału w matematycznej teorii optymalizacji. Whitney, znany z wprowadzenia teorii grafów w matematykę, mógł nie być bezpośrednio związany z problemem komiwojażera, ale jego prace niewątpliwie wpłynęły na badania nad problemem komiwojażera, rozwijane później przez Flooda oraz innych naukowców m.in. George'a Dantziga, który opracował słynny algorytm sympleksowy służący rozwiązywaniu złożonych problemów optymalizacyjnych.

Jednym z bardziej znanych przykładów wczesnego podejścia do optymalizacji tras, które z czasem przekształciło się w badania nad problemem komiwojażera, było doświadczenie Henry'ego Cleveland z Page Seed Company z 1925 roku. Wspomniany sprzedawca zbierający zamówienia na produkty rolnicze, takie jak kukurydza, był niezadowolony z istniejących planów tras i zdecydował się na wprowadzenie własnych ulepszeń. Jego zoptymalizowana trasa

obejmowała 350 przystanków w stanie Maine, rozpoczynając się w Kittery i kończąc w pobliżu Springvale, co zostało przedstawione na rysunku 9. Przebieg tej trasy, który odbył się od 9 lipca do 24 sierpnia 1925 roku, jest uznawany za jedno z pierwszych praktycznych zastosowań koncepcji optymalizacji trasy, będącej zalążkiem późniejszego problemu komiwojażera.



Rysunek 9. Zoptymalizowana trasa podróży po stanie Maine z 1925 roku.
[źródło: (Cook, 2012)]

Opisany przykład ukazał, jak innowacyjne podejście Cleveland'a do planowania tras mogło stanowić prototyp dla metod, które później znalazłyby zastosowanie w bardziej skomplikowanych algorytmach komiwojażera. Użycie przez niego map, pinezek i sznurków do wizualizacji i planowania trasy było swoistym analogowym narzędziem, które przewidziało późniejsze cyfrowe narzędzia optymalizacji.

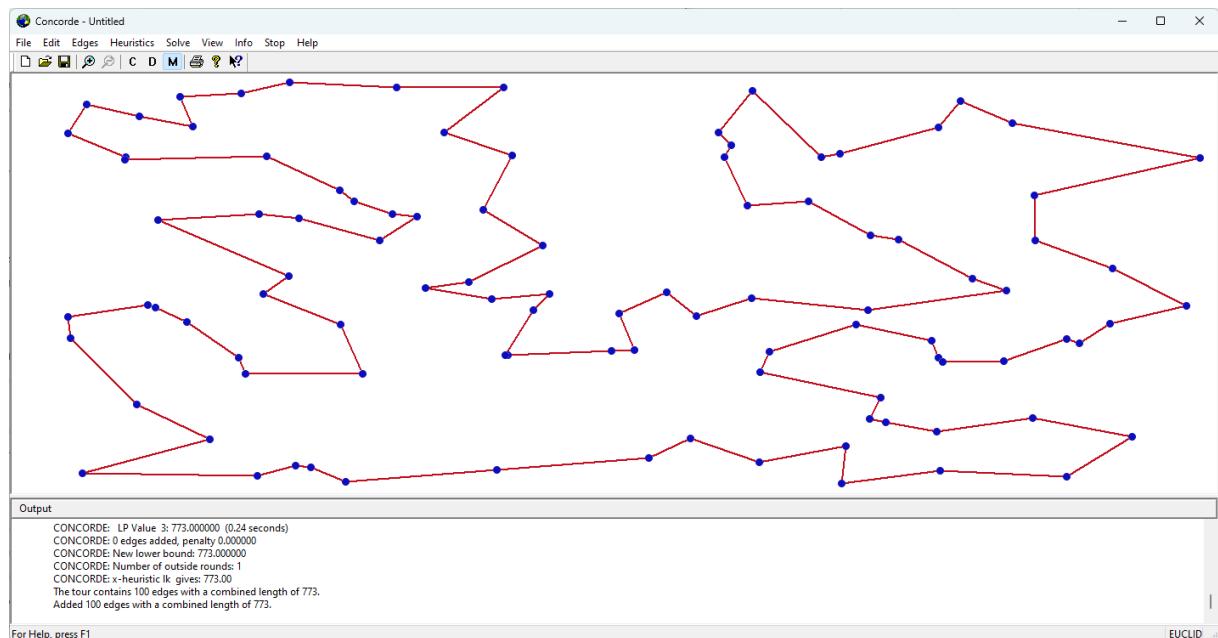
Prace George'a Dantziga, Raya Fulkersona i Selmera Johnsona na przełomie lat czterdziestych i pięćdziesiątych XX wieku przyniosły rewolucję w sposobie rozwiązywania problemów komiwojażera dzięki zastosowaniu nowoczesnych algorytmów matematycznych. (Kowalik, 2011) W 1949 roku Dantzig wprowadził do programowania liniowego algorytm simplex, co zmieniło sposób rozwiązywania złożonych problemów optymalizacyjnych. Algorytm ten pozwalał na efektywne znalezienie najlepszego rozwiązania wśród wielu

możliwych opcji, co było kluczowe w kontekście zastosowań w planowaniu i zarządzaniu, a także w problemie komiwojażera, gdzie trzeba wybrać najkrótszą możliwą trasę spośród wielu kombinacji. Wraz z Fulkersonem i Johnsonem w 1954 roku, opracował algorytm, który znaczaco przyspieszył rozwiązywanie dużych instancji problemów sieciowych, w tym problemu komiwojażera. Ich prace były oparte na metodach dekompozycji, które umożliwiały rozkładanie dużych problemów na mniejsze, łatwiejsze do zarządzania fragmenty. Dało to możliwość do rozwiązywania skomplikowanych problemów logistycznych związanych z optymalizacją tras oraz było fundamentem dla kolejnych pokoleń systemów informatycznych zajmujących się optymalizacją. Mapy, pineski oraz sznurki zamieniono na skomplikowane algorytmy zdolne do zarządzania i optymalizacji tras w dynamicznie zmieniającym się świecie globalnego handlu i logistyki.

Przełomowe badania Danzinga, Fulkersona i Johnsona spowodowały, że świat nauki zaczął poszukiwać kolejnych możliwości usprawnienia podejścia do problemu komiwojażera. Pomimo że algorytm simplex, jak i metody dekompozycji diametralnie przyspieszyły możliwość rozwiązywania złożonych problemów logistycznych, to wciąż poszukiwane były różnego rodzaju metody, które byłyby zdolne do radzenia sobie z jeszcze większymi instancjami problemu, których złożoność wykraczała poza możliwości nawet najbardziej zaawansowanych technik eklektycznych. Takie zapotrzebowanie sprawiło, że znaczenia zaczęły nabierać algorytmy aproksymacyjne i heurystyczne, które umożliwiały znalezienie rozwiązań dobrych, choć nie zawsze optymalnych, w sposób znacznie szybszy niż metody tradycyjne. (Wikipedia, 2023) Algorytm aproksymacyjny to algorytm znajdujący rozwiązanie problemu optymalizacyjnego, który jest bliski rozwiązaniu optymalnemu, lecz wygenerowane w sposób znacznie szybszy niż przy użyciu dokładnych metod. Algorytmy te stosowane są do problemów, dla których nieznane są algorytmy dokładne m.in. do problemu komiwojażera oraz innych problemów NP-zupełnych. Z kolei algorytmy heurystyczne (Encyklopedia Algorytmów, 2020) to metody obliczeniowe stosowane w celu rozwiązywania problemów optymalizacyjnych, gdy problem jest NP-trudny, a znalezienie dokładnego rozwiązania w rozsądny czasie jest niepraktyczne lub niemożliwe, jednak nie zawsze prowadzą do optymalnego rozwiązania, ale są na ogół wystarczająco dobre w kontekście praktycznych zastosowań. Przykładami algorytmów heurystycznych związanych bezpośrednio z problemem komiwojażera jest algorytm najbliższego sąsiada, algorytmy genetyczne, algorytm mrówkowy. Do kluczowych cech algorytmów heurystycznych zalicza się szybkość działania, przybliżenie oraz elastyczność. Heurystyki są zaprojektowane tak, aby szybko generować rozwiązania, nawet dla bardzo dużych i złożonych problemów.

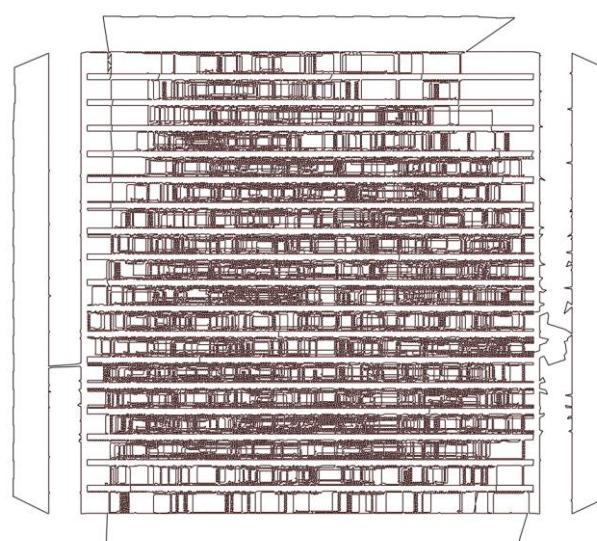
Algorytmy heurystyczne zwracają rozwiązania, które są bliskie optymalnym, choć nie zawsze są one dokładne, a dzięki zastosowaniu elastyczności mogą być łatwo dostosowywane do konkretnych problemów i zmieniających się warunków.

W erze cyfryzacji zaawansowane obliczenia zaczęły być wykonywane z pomocą wszechobecnego sprzętu komputerowego. Zaczęto opracowywać różnego rodzaju oprogramowania oraz narzędzia, które miały za zadanie sprawdzić, czy teoretyczne koncepcje znajdują zastosowanie w praktyce. Jednym z bardziej znanych przykładów oprogramowania specjalistycznego wykorzystującego algorytmy heurystyczne do rozwiązywania problemu komiwojażera jest Concorde TSP Solver autorstwa Davida Applegate, Roberta Bixby, Václava Chvátala oraz Williama Johna Cooka. (Wikipedia, 2023) Program Concorde z pomocą różnych technik heurystycznych i dokładnych daje możliwość efektywnego rozwiązywania problemu komiwojażera. Oprogramowanie umożliwiało odnalezienie najkrótszej trasy, która pozwalała odwiedzić listę miast i wrócić do punktu wyjścia, odwiedzając każde miasto tylko raz. Dodatkowo był w stanie znaleźć trasę, która jest nie tylko teoretycznie optymalna, ale również praktycznie użyteczna w realnych scenariuszach. Na rysunku 10 przedstawiono interfejs użytkownika programu Concorde. Program ten przedstawia graficznie trasę komiwojażera jako sieć połączonych wierzchołków i krawędzi.



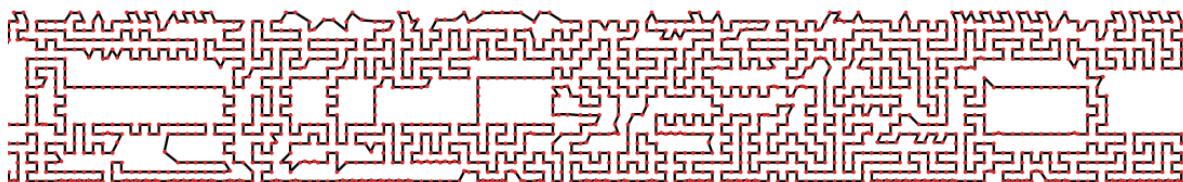
Rysunek 10. Zrzut ekranu z programu Concorde.
[źródło: opracowanie własne]

Niebieskie punkty reprezentowały miasta lub punkty, które komiwojażer musiał odwiedzić, z kolei czerwone linie przedstawiały trasę łączącą te miasta w kolejności, w jakiej powinny być odwiedzane, aby suma przebytych odległości była jak najmniejsza. Na powyższym przykładzie obliczona została trasa dla trasy najbliższego sąsiada, której łączna długość wynosi 1011 jednostek. Następnie program przechodził do zaawansowanej optymalizacji za pomocą technik programowania liniowego i algorytmów cięcia, które systematycznie redukowały długość trasy przez iteracyjne dodawanie ograniczeń matematycznych. Każda kolejna iteracja przynosiła poprawę rozwiązania, gdzie długość trasy były sukcesywnie zmniejszane, aż do osiągnięcia znacznie krótszej trasy wynoszącej 773 jednostki. Przedstawione oprogramowanie ukazało ewolucję w dziedzinie algorytmów, przekształcając teoretyczne koncepcje w nowoczesne potężne aplikacje, z pomocą których użytkownicy mogą w efektywny sposób spróbować swoich sił z jednymi z najbardziej złożonych wyzwań optymalizacyjnych spotykanych w praktyce. (Wikipedia, 2024) Program Concorde został wykorzystany w maju 2004 roku do wyznaczenia najkrótszej trasy dla wszystkich 24 978 miast w Szwecji, czego rezultatem było otrzymanie trasy mierzącej 72 500 kilometrów. Udowodniono również, że krótsza droga nie istnieje. W 2005 ustanowiono kolejny rekord, tym razem polegał on na odwiedzeniu 33 810 punktów na płytce drukowanej za pomocą programu Concorde i oszacowano, że trasa wyniosła 66 048 945 jednostek oraz ponownie udowodniono, że krótsza droga nie istnieje. (Traveling Salesman Problem, 2015) Z kolei w 2006 rozwiązano problem komiwojażera dla 85 900 punktów co stanowiło ówcześnie rekord świata. Rysunek 11 zobrazował rozwiązanie problemu komiwojażera dla 85900 miast z zastosowaniem chipa komputerowego.



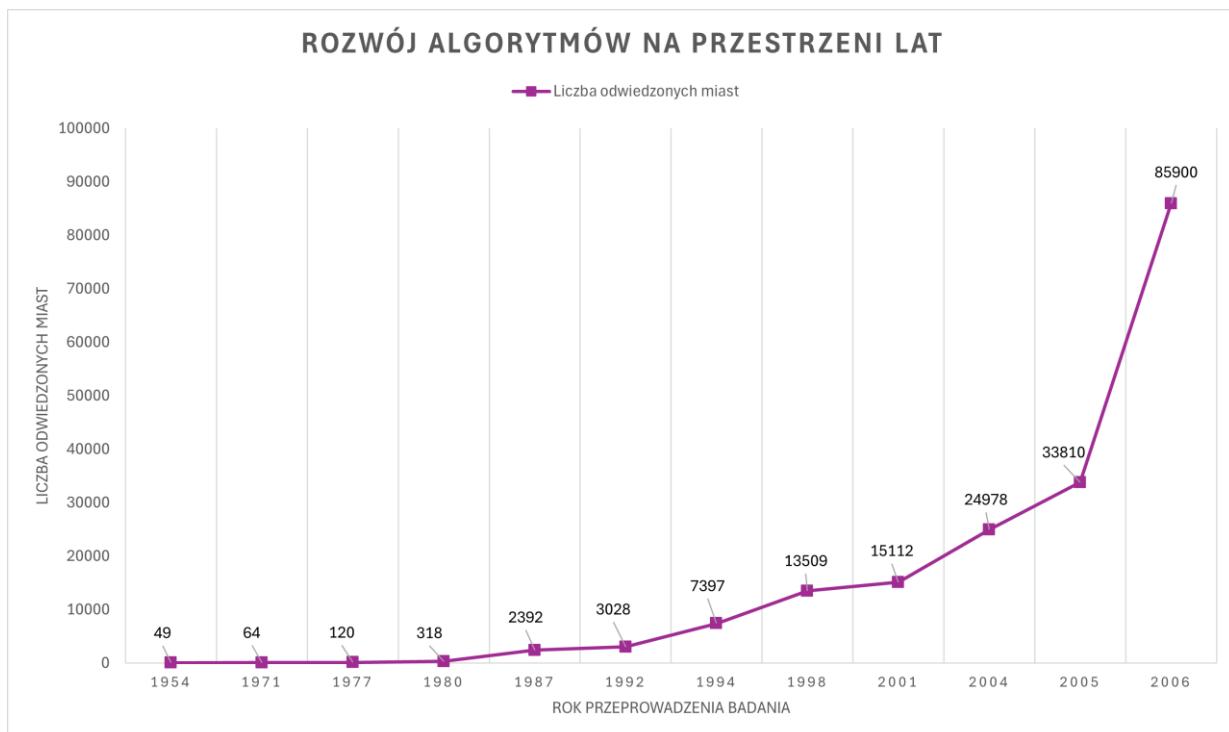
Rysunek 11. Rozwiązanie TSP dla 85 900 miast w zastosowaniu chipa komputerowego.
[źródło: (Cook, 2012)]

Z kolei rysunek 12 przedstawia zbliżenie małego obszaru części trasy obejmującej 85 900 miast w sposób prostszy do obserwacji.



Rysunek 12. Przedstawienie małego obszaru trasy obejmującej 85 900 iast
[źródło: (Cook, 2012)]

Rysunek 12 prezentuje rozwój algorytmów na przestrzeni lat. (Cook, 2012) Wykres rozpoczął się od ręcznie opracowanego przez Danzinga w 1954 roku przykładu dla 49 miast, aż do ostatniego rozwiązania z 2006 roku.



Rysunek 13. Rozwój algorytmów na przestrzeni lat
[źródło: opracowanie własne na podstawie (Cook, 2012)]

(World Traveling Salesman Problem, 2021) W 2007 roku duński informatyk Keld Helsgaun z wykorzystaniem programu Concorde w połączeniu z solverem³ programowania CPLEX⁴ ustanowił nowy wynik najlepszego dolnego ograniczenia długości trasy, która

³ Solver – narzędzie komputerowe służące do automatycznego znajdowania rozwiązań złożonych problemów matematycznych.

⁴ CPLEX – zaawansowane oprogramowanie do optymalizacji liniowej i całkowitoliczbowej.

wyniosła 7 512 218 268 metrów, co uświadamia, że trasa Helsguna jest zaledwie o 0,0471% dłuższa niż optymalna trasa. W 2021 roku Keld Helsgaun wykorzystując własny algorytm heurystyczny LKH osiągnął nowy rekord długości trasy, który wyniósł 7 515 755 956 metrów. (Strąk, 2017) Algorytm heurystyczny Lin-Kernighan-Heuristic jest jednym z najbardziej efektowych algorytmów heurystycznych do rozwiązywania problemu komiwojażera. Jest to ulepszona wersja algorytmu Lin-Kernighan⁵ zawierająca techniki opracowane przez Kelda Helsgauna, które poprawiają wydajność oraz jakość znajdowanych rozwiązań. LKH wykorzystuje metody lokalnego przeszukiwania, aby iteracyjnie poprawiać istniejące rozwiązanie, zamieniając segmenty trasy w taki sposób, aby uzyskać jak najkrótszą możliwą ścieżkę łączącą wszystkie punkty.

Problem komiwojażera mimo swojej pozornej prostoty znajduje szerokie zastosowanie w różnych dziedzinach, co potęguje jego znaczenie zarówno w teorii, jak i w praktyce. Przede wszystkim problem ten wykorzystywany jest w logistyce oraz planowaniu tras, gdzie pomaga w optymalizacji tras dostaw, minimalizując czas podróży i koszty paliwa. Wykorzystywany jest również między innymi w bioinformatyce do sekwencjonowania DNA, gdzie kolejność fragmentów DNA musi być ułożona w możliwie najbardziej efektywny sposób. Problem komiwojażera obecny jest również w dziedzinie robotyki, na przykład w przypadku automatycznych magazynów oraz systemach produkcji. We wspomnianych sektorach roboty wykorzystują algorytmy bazujące na problemie TSP do optymalizacji ścieżek ruchu, w celu wykonywania zadań efektywnie i z minimalnym zużyciem energii. Wspomniane przykłady stanowią tylko część przykładowych praktycznych zastosowań. W związku z ciągle rozwijającą się technologią, szczególnie w dziedzinie uczenia maszynowego, a także sztucznej inteligencji istnieje szansa na powstanie nowych metod rozwiązywania tego NP-trudnego problemu. Algorytmy uczenia maszynowego mogą być wykorzystywane do tworzenia bardziej efektywnych heurystyk, które mogą nauczyć się wzorców i optymalizować decyzje w bardziej dynamicznych i nieprzewidywalnych środowiskach. Jednym z głównych wyzwań na przyszłość jest poprawa skalowalności algorytmów do bardzo dużych instancji problemów. Istnieje również możliwość, że współpraca pomiędzy różnymi dziedzinami nauki może dać możliwość wprowadzenia nowych zastosowań problemu komiwojażera.

⁵ Algorytm Lin-Kernighan to heurystyka do rozwiązywania problemu komiwojażera, opierająca się na metodzie lokalnego przeszukiwania w celu poprawy istniejącej trasy przez zamianę segmentów.

1.2. Opis problemu

(Sysło, Deo i Kowalik, 1995) Problem komiwojażera stanowi klasyczny problem optymalizacji kombinatorycznej. Polega na odnalezieniu najkrótszej możliwej drogi, w której istnieje możliwość odwiedzenia wszystkich dostępnych miast wyłącznie jeden raz, powracając na samym końcu do punktu startowego. (Cormen, Leiserson, Rivest i Stein, Wprowadzenie do algorytmów, 2018) Problem TSP ma wiele zastosowań praktycznych np. w logistyce do planowania tras flot pojazdów, czy też planowania wycieczek. Stosowany jest również przy optymalizacji ścieżek narzędzi na liniach produkcyjnych w dziedzinie produkcji, czy też po pewnych modyfikacjach do sekwencjonowania DNA, gdzie opisane miasto zastępuje się fragmentami DNA, natomiast odległość zastąpiona jest funkcją podobieństwa między poszczególnymi fragmentami materiału genetycznego DNA.

(Sysło, Deo i Kowalik, 1995) W wariantie decyzyjnym problemu komiwojażera dany jest nieskierowany graf pełny posiadający nieujemne i całkowite wagi krawędzi. Graf ten można przedstawić jako macierz wag $W = [w_{ij}]$. W tym przypadku w_{ij} symbolizują wagę krawędzi pomiędzy wierzchołkami i oraz j . (Cormen, Leiserson, Rivest i Stein, Wprowadzenie do algorytmów, 2007) Problem komiwojażera stanowi klasyczny przykład problemu NP-trudnego. Certyfikat w problemie decyzyjnym TSP składa się z wierzchołków cyklu, wymienionych po kolej. Za pomocą czasu wielomianowego można w prosty sposób zweryfikować czy krawędziami danego cyklu istnieje możliwość dotarcia do wszystkich wierzchołków oraz czy ich waga łącznie wynosi k bądź mniej. Fakt, że problem komiwojażera jest NP-trudny można zaprezentować za pomocą redukcji problemu cyklu Hamiltona, gdzie posiadając na wejściu graf G należy skonstruować graf pełny G' , który posiada te same wierzchołki co poprzednik. Kolejny krok stanowi nadanie krawędzią (u,v) w nowo utworzonym grafie wagi 0 w przypadku, gdy (u,v) są krawędzią w grafie G lub wagi 1 jeżeli w poprzedniku taka krawędź nie występuje.

Dla przykładu można założyć, że dane jest pięć miast A, B, C, D oraz E. Odległości pomiędzy nimi przedstawione zostały w tabeli 1. Kolorem błękitnym zaznaczone zostały odległości pomiędzy poszczególnymi miastami.

Tabela 1.

Odległości pomiędzy miastami A, B, C, D, E.

	A	B	C	D	E
A	-	2	9	10	7
B	2	-	6	4	3
C	9	6	-	8	3
D	10	4	8	-	1
E	7	3	3	1	-

źródło: opracowanie własne

W tym przykładzie należy odnaleźć najkrótszą trasę rozpoczynającą się w jednym z miast, odwiedzając przy tym wszystkie możliwe miasta wyłącznie jeden raz, wracając na koniec do miasta, w którym podróż została rozpoczęta. Jednym z możliwych rozwiązań jest trasa $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow A$, której całkowita długość wynosi $2 + 4 + 1 + 3 + 9 = 19$.

Problem komiwojażera rozróżniany jest jako dwie klasy problemu:

- symetryczny problem komiwojażera;
- asymetryczny problem komiwojażera.

(Wikipedia, 2024) Symetryczny problem komiwojażera (STSP – Symmetric Travelling salesman problem) to wariant klasycznego problemu TSP. Klasa ta cechuje się tym, że odległości pomiędzy dowolnymi miastami A i B są identyczne w obydwie strony. Macierz odległości jest symetryczna.

(Wikipedia, 2024) Asymetryczny problem komiwojażera (ATSP – Asymmetric Travelling salesman problem) jest to druga klasa problemu TSP, która charakteryzuje się tym, że odległość pomiędzy miastem A oraz B jest inna niżeli z miasta B do miasta A.

Sporym wyzwaniem dla problemu komiwojażera stanowi bardzo duża ilość danych, które należy poddać analizie. (Wikipedia, 2024) Odległość w TSP pełni zdecydowanie kluczową rolę przez co można go podzielić na problem metryczny oraz problem niometryczny. Problem metryczny stanowi wariant, w którym odległości pomiędzy miastami muszą spełniać nierówność trójkąta. Nierówność trójkąta głosi, że dla trzech dowolnych miast A, B oraz C, odległość pomiędzy miastem A i miastem C jest mniejsza lub równa sumie odległości pomiędzy miastem A oraz B oraz B i C. Formalnie za pomocą wzoru można zapisać to w następujący sposób:

$$d(A, C) \leq d(A, B) + d(B, C) \quad (7)$$

W powyżej przedstawionym wzorze (7) można zaobserwować, że nierówność trójkąta została spełniona oraz odległości $d(A, B) = d(B, A)$ są symetryczne.

Z kolei w przypadku problemu niemetrycznego odległości pomiędzy miastami nie mają narzuconego warunku spełniania nierówności trójkąta tj. może istnieć przypadek, gdzie odległość bezpośrednia między danymi dwoma miastami jest większa niż suma odległości przez inne możliwe miasto.

Rozważając możliwe koszty przejazdu możliwymi środkami transportu np. koleją czy też z pomocą linii lotniczych, może wystąpić sytuacja, w której koszt przejazdu koleją odwiedzając przy tym dodatkowe miasta może być niższy niżeli bezpośredni lot samolotem. Z tego powodu można wyróżnić następujące rodzaje metrycznych problemów komiwojażera:

- Metryka euklidesowa;
- Metryka Manhattan.

(Wikipedia, 2024) Metryka Euklidesowa stanowi klasyczną miarę odległości pomiędzy dwoma punktami w przestrzeni euklidesowej. W odniesieniu do problemu komiwojażera metryka ta stosowana jest w celu obliczenia rzeczywistej odległości między miastami na płaszczyźnie. Dla dwóch punktów $A = (x_1, y_1)$ oraz $B = (x_2, y_2)$, odległość euklidesowa d określana jest wzorem:

$$d(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (8)$$

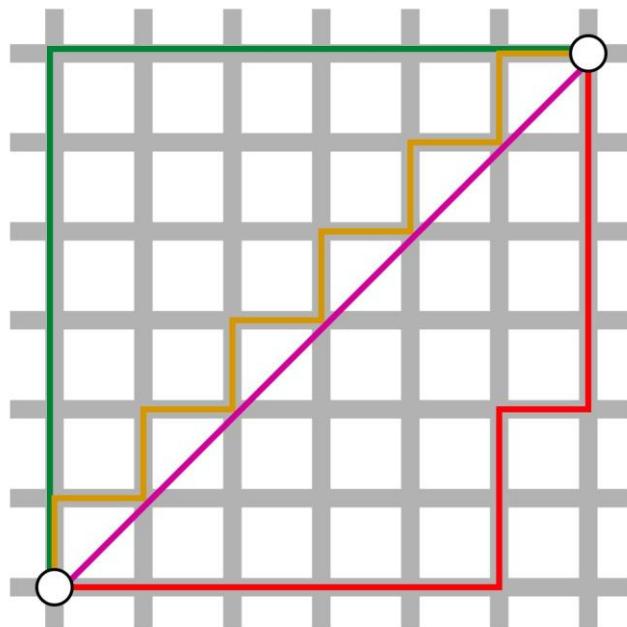
(Wikipedia, 2024) Metryka Manhattan, nazywana również odlegością miejską bądź siatką taksówkarską odpowiada za pomiar odległości między dwoma punktami rozmieszczonymi w przestrzeni jako suma bezwzględnych różnic ich współrzędnych. W przypadku problemu komiwojażera wykorzystana została do obliczeń odległości w przypadkach, gdzie ruch odbywa się wzdłuż prostych osi, tak jak w siatce miejskiej. Dla punktów $A = (x_1, y_1)$ oraz $B = (x_2, y_2)$, odległość Manhattan d wyznacza się za pomocą ogólnego wzoru:

$$d(A, B) = \sum_{i=1}^d |x_{i2} - x_{i1}| \quad (9)$$

Odległość Manhattan można również przedstawić jako dwuwymiarową przestrzeń za pomocą wzoru:

$$d(A, B) = |x_2 - x_1| + |y_2 - y_1| \quad (10)$$

(abcdef.wiki, 2024) W kontekście problemu komiwojażera, w którym odległość pomiędzy punktami wyliczana jest za pomocą metryki Manhattan, trasa minimalizuje odległość wzduż osi poziomych oraz pionowych. Cechą ta jest korzystna w środowiskach miejskich, w których przemieszczanie się pomiędzy punktami odbywa się po ulicach tworzących siatkę. Jako ciekawostkę warto dodać, że nazwa metryki miejskiej tzw. metryki Manhattan wywodzi się od nowojorskich taksówek poruszających się po ulicach miasta oraz wykorzystywana była do zminimalizowania czasu przejazdu stosując ograniczenie w ruchu w kierunkachpółnoc-południe oraz wschód-zachód. Z kolei w przypadku obliczeń odległości za pomocą metryki euklidesowej, trasa dokonuje minimalizacji rzeczywistych odległości w przestrzeni dwuwymiarowej. Na rysunku 14 przedstawione zostało porównanie omówionych metryk. Kolorem fioletowym oznaczona została metryka euklidesowa, natomiast pozostałymi kolorami tj. zielonym, pomarańczowym oraz czerwonym zaznaczona została metryka miejska.



Rysunek 14. Porównanie metryki Manhattan oraz metryki euklidesowej
[źródło: opracowanie własne na podstawie (Wikipedia, 2024)]

Metryki Manhattan zgodnie z założeniami mają takie same odległości, z kolei metryka euklidesowa oznaczona na fioletowo posiada najkrótszą długość.

1.3. Rodzaje problemów

(Wikipedia, 2024) Problemy obliczeniowe klasyfikuje się w różnego rodzaju klasy złożoności. Klasy te charakteryzują się poprzez trudność rozwiązywania problemów pod względem różnego rodzaju zasobów m.in. czasu bądź pamięci niezbędnej do rozwiązywania tych problemów. Rozważany w tej pracy problem komiwojażera zalicza się do klasy NP-trudnych. Oznacza to, że jest co najmniej tak trudny, jak najtrudniejsze problemy w tej klasie oraz nie istnieje algorytm działający w czasie wielomianowym, który znalazłby optymalne rozwiązanie problemu komiwojażera. (Brilliant.org, 2024) NP-trudność uświadamia, że dany problem nie należy do klasy NP, charakteryzującej się występowaniem problemów, w których można szybko sprawdzić poprawność rozwiązania, lecz każdy problem klasy NP z pomocą transformacji działającej w czasie wielomianowym można do niego sprowadzić.

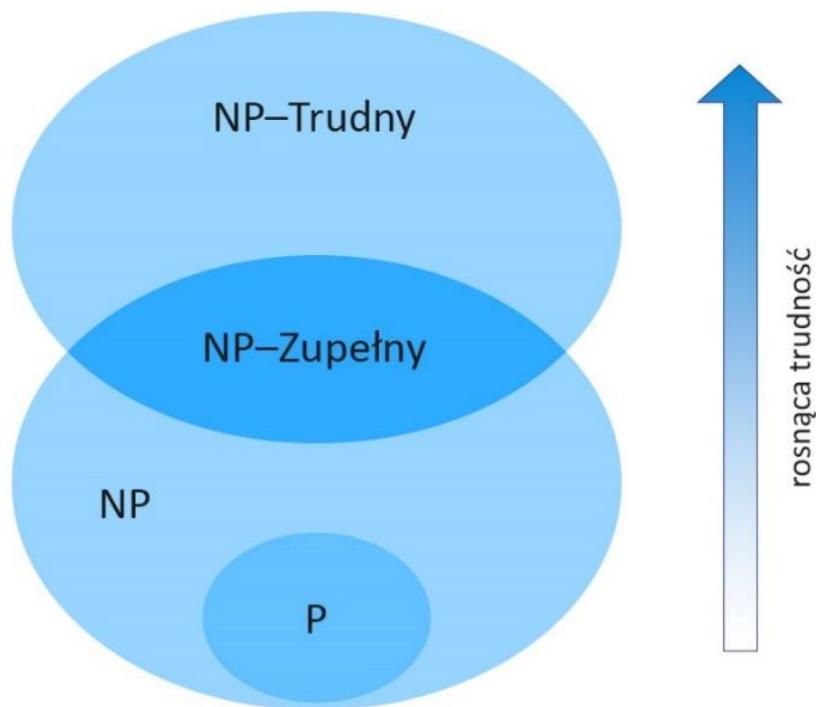
(Ladner, 1975) Istnieje wiele klas złożoności problemów obliczeniowych. Zalicza się do nich do nich następujące klasy:

- Klasa P (Polynomial) – najmniejsza klasa problemów, które można rozwiązać w czasie wielomianowym poprzez deterministyczną maszynę Turinga⁶;
- Klasa NP (Nondeterministic Polynomial time) – Klasa ta zawiera problemy decyzyjne, które można rozwiązać w czasie wielomianowym z pomocą nietrivialnej maszyny Turinga, bądź równoważnie, których rozwiązania mogą być zweryfikowane w czasie wielomianowym przez deterministyczną maszynę;
- Klasa NP-Zupełne (Nondeterministic Polynomial Complete) – stanowi podzbiór klasy NP, w którym umiejscowione zostały najtrudniejsze problemy klasy złożoności obliczeniowej, ze względu na brak możliwości odnalezienia rozwiązania problemu w czasie wielomianowym;
- Klasa NP-Trudne (Nondeterministic Polynomial Hard) – stanowi zbiór problemów co najmniej tak trudnych, jak najtrudniejsze problemy klasy NP. W odróżnieniu od problemów klasy Nondeterministic Polynomial nie są to koniecznie problemy decyzyjne, ponieważ mogą być trudniejsze niż NP, oraz obejmować problemy spoza NP.

Na rysunku 15 zaprezentowana została hierarchia problemów decyzyjnych w teorii złożoności obliczeniowej, ukazując względne położenie i związki pomiędzy klasami

⁶ Maszyna Turinga – teoretyczny model obliczeniowy używany do opisu algorytmów, składający się z nieskończonie długiej taśmy, głowicy odczytującej, zapisującej oraz zestawu instrukcji stresujących działaniem głowicy.

problemów. Strzałka po prawej stronie obrazuje wzrastającą trudność problemów w ramach przedstawionych klas od najłatwiejszej P, poprzez NP, aż do najtrudniejszych NP-Trudnych.



Rysunek 15. Hierarchia problemów decyzyjnych
[źródło: opracowanie własne na podstawie (Brilliant.org, 2024)]

1.4. Podstawowe pojęcia

Podrozdział skupia swoją uwagę na przedstawieniu podstawowych pojęć związanych z problemem komiwojażera, które warto znać w celu głębszego zrozumienia zagadnienia.

1.4.1. Graf

(Wojciechowski i Pieńkocz, 2013) Graf zdefiniowano jako uporządkowaną parę zbioru wierzchołków V oraz krawędzi E , wykorzystywaną do badania relacji między obiektami. Określono, że graf musi posiadać przynajmniej jeden wierzchołek $V \neq \emptyset$, a także przecięcie zbiorów wierzchołków V oraz zbioru krawędzi E jest w grafie zbiorem pustym $V \cap E = \emptyset$. Stwierdzono, że ma żadnych elementów wspólnych pomiędzy wierzchołkami a krawędziami

grafu, co jest zgodne z definicją grafu, w którym wierzchołki i krawędzie są traktowane jako oddzielne, niezależne zbiory. Krawędź grafu $e \in E$ stanowi uporządkowana para wierzchołków $e = (v_1, v_2)$, $v_1 \in V$, $v_2 \in V$. (Wilson, 2000) W najprostszych słowach, graf określany jest jako zbiór punktów, które łączy się liniami tak, że każda linia zaczynała się i kończyła na jednym z punktów. (Wojciechowski i Pieńkocz, 2013) W przypadku, gdy w zbiorze krawędzi występują powtarzające się elementy, krawędzie nazwano równoległyimi lub wielokrotnymi. Liczbę wierzchołków grafu oznaczono jako n , natomiast liczbę krawędzi jako q . Zapis ten wykorzystano do klasyfikacji grafów skończonych, w których n oraz q są liczbami skończonymi. W przypadku, gdy $(v_1, v_2) \in E$ oraz $(v_2, v_1) \in E$ krawędź nazwano krawędzią nieorientowaną, zwaną również nieskierowaną i oznaczano symbolem $e = \{v_1, v_2\}$. W przypadku, gdy $(v_1, v_2) \in E$ oraz $(v_2, v_1) \notin E$, krawędź nazwano łukiem, krawędzią zorientowaną lub skierowaną. (Wilson, 2000) Nadano poszczególnym wierzchołkom numerację, a także uznano je za reprezentację obiektów, natomiast krawędzie uznano za relacje pomiędzy nimi. W przypadku, gdy krawędzie posiadają kierunki, mówiono o grafie skierowanym. Krawędzie mogły posiadać wagi reprezentujące m.in. odległość pomiędzy poszczególnymi wierzchołkami. (Wikipedia, 2024) Według dostępnych informacji pierwszym badaczem grafów uznano Leonarda Eulera, który wykorzystał grafy w swojej teorii dotyczącej mostów królewieckich, natomiast wprowadzenie słowa „graf” przypisano Jamesowi Josephowi Sylvesterowi w 1878 roku.

1.4.2. Cykl Eurela

(Weisstein, 2024) Cykl Eurela stanowi trasę w grafie rozpoczynającą się i kończącą na tym samym wierzchołku, przekraczając każdą z krawędzi wyłącznie raz. Pojęcie cyklu Eurela wynika z pracy autora nad problemem mostów królewieckich, co zostało przedstawione w genezie problemu. Tym samym udowodnił, że aby graf mógł posiadać cykl Eurela, każdy z jego wierzchołków powinien posiadać parzystą ilość krawędzi. W przypadku stopnia parzystego graf spełnia warunek przejścia przez każdą z krawędzi wyłącznie raz, bez zatrzymania się w jakimkolwiek wierzchołku. Atutem cyklu Eurela jest fakt, że żadna z krawędzi grafu nie jest pomijana, obejmując w ten sposób cały graf powracając do punktu startowego, co stanowi kluczową cechę m.in. w aplikacjach do projektowania efektywnych tras, bez konieczności powtarzania tych samych tras.

1.4.3. Twierdzenie Eurela

(Wilson, 2000) Twierdzenie Eurela obwieszcza, że dla każdego spójnego grafu planarnego, oznaczonego symbolem G , który można zilustrować na płaszczyźnie bez przecinających się krawędzi, spełniona zostaje równość:

$$V - E + F = 2 \quad (11)$$

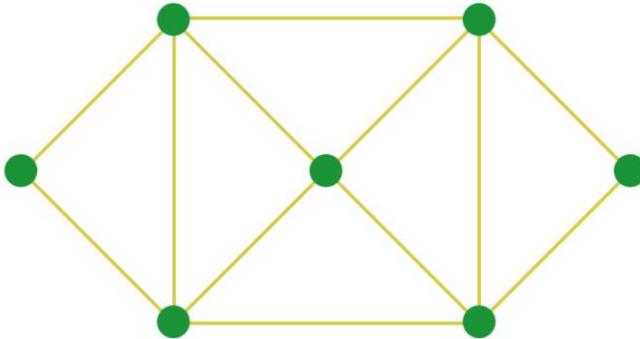
Gdzie:

- V oznacza liczbę wierzchołków grafu G
- E oznacza liczbę krawędzi grafu G
- F oznacza liczbę ścian grafu G , włączając w to ścianę zewnętrzną

Dowód twierdzenia wykorzystuje indukcję matematyczną względem liczby wierzchołków bądź krawędzi w grafie. W podstawowym założeniu dla grafów z jednym wierzchołkiem bez krawędzi $E = 0$, ściana zewnętrzna jest jedyną ścianą, a co za tym idzie równość zostaje spełniona. Kolejny krok to rozważenie przypadków w sytuacji, gdy dodane zostały kolejne krawędzie doprowadzając do analizy ilości ścian. W przypadku, gdy graf jest acyklicznym grafem spójnym, czyli innymi słowy drzewem to równość także zachodzi, ze względu na to, że drzewo z n wierzchołkami ma zawsze $n - 1$ krawędzi, a także jedną ścianę zewnętrzną. Refleksje na temat grafów nie będącymi drzewami obejmują schematy, gdzie grafy zawierają cykle, a co za tym idzie dodatkowe ściany wewnętrzne. W takich sytuacjach usunięcie krawędzi z cyklu doprowadza do zmniejszenia liczby ścian o jedną, dzięki czemu równość zarówno w przypadku dodatnia, jak i usunięcia krawędzi zostaje zachowana.

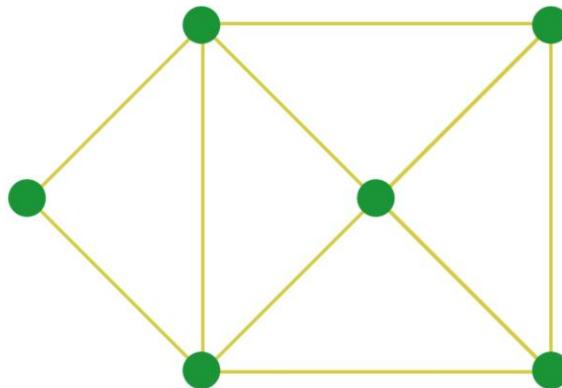
1.4.4. Graf eurelowski i półeurelowski

(Wilson, 2000) Graf eurelowski jest to spójny graf, w którym istnieje możliwość odnalezienia cyklu Eurela. Proces ten stanowi zamkniętą trasę przechodzącą przez każdą krawędź grafu wyłącznie jeden raz, rozpoczynając się i kończąc w tym samym punkcie, co stanowi kluczowy wymóg cyklu Eurela. Kluczowym warunkiem grafu eulerowskiego jest stopień wierzchołków w grafie, który musi być parzysty. Oznacza to, że do każdego z wierzchołków musi wchodzić parzysta ilość krawędzi. Przykładowy graf eurelowski przedstawiony został na rysunku 16.



Rysunek 16. Graf eurelowski
 [źródło: opracowanie własne na podstawie (Wilson, 2000)]

W przeciwnym przypadku jest to graf półeurelowski, który zezwala na istnienie dwóch wierzchołków o stopniu nieparzystym, a co za tym idzie istnieje możliwość rozpoczęcia i zakończenia ścieżki w dwóch różnych wierzchołkach, przechodząc przez każdą z krawędzi wyłącznie raz. Przykład takiego grafu został przedstawiony na rysunku 17.



Rysunek 17. Graf półeurelowski
 [źródło: opracowanie własne na podstawie (Wilson, 2000)]

1.4.5. Cykl Hamiltona

(Cormen, Leiserson, Rivest i Stein, Wprowadzenie do algorytmów, 2018) Cyklem Hamiltona określono ścieżkę w grafie, która przechodzi przez wszystkie wierzchołki grafu wyłącznie jeden raz, rozpoczynając i kończąc w tym samym punkcie, tworząc w ten sposób zamknięty cykl. Formułując cykl Hamiltona matematycznie dla grafu $G = (V, E)$ zawierającym zbiór wierzchołków V oraz zbiór krawędzi E , cykl opisuje się jako permutację wierzchołków v_1, v_2, \dots, v_n w następujący sposób:

$$(v_i, v_{i+1}) \in E \text{ dla wszystkich } i = 1, 2, \dots, n - 1, \quad (12)$$

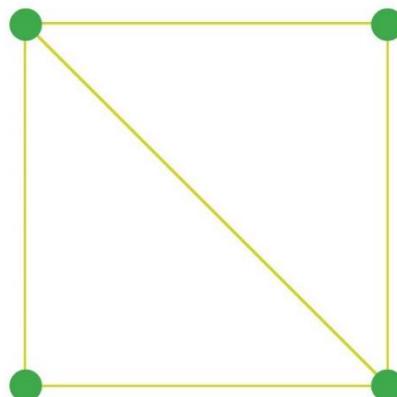
$$(v_n, v_1) \in E, \text{ zamykając cykl} \quad (13)$$

W równaniu (12) wzór informuje, że dla każdego wierzchołka v_i istnieje krawędź, która łączy go bezpośrednio z następnym wierzchołkiem v_{i+1} w sekwencji cyklu. Oznacza to, że z każdego wierzchołka cyklu istnieje możliwość bezpośredniego przejścia, aż do przedostatniego. Równanie (13) odnosi się do ostatniego w cyklu wierzchołka v_n , a także do pierwszego wierzchołka v_1 . Cykl zostanie zamknięty tylko wtedy, kiedy pomiędzy tymi wierzchołkami istnieje krawędź. W sytuacji, gdy wszystkie wierzchołki zostaną odwiedzone, ostatni musi być bezpośrednio połączony z pierwszym wierzchołkiem, tworząc zamkniętą pętlę.

(Wojciechowski i Pieńkocz, 2013) W przypadku grafu niezorientowanego, cykl Hamiltona przechodzi przez wszystkie wierzchołki wyłącznie raz, a na końcu powraca do punktu początkowego. Z kolei w grafie zorientowanym ścieżka przechodzi przez skierowane krawędzie, zachowując kierunkowość grafu.

1.4.6. Graf Hamiltonowski

(Wilson, 2000) Graf Hamiltonowski to graf spójny, w którym istnieje cykl hamiltonowski przechodzący przez każdy z wierzchołków wyłącznie jeden raz. Cykl rozpoczyna i kończy bieg w tym samym wierzchołku. Na rysunku 18 znajdującym się poniżej przedstawiony został przykładowy graf hamiltonowski.



Rysunek 18. Graf hamiltonowski
[źródło: opracowanie własne na podstawie (Wilson, 2000)]

(Wikipedia, 2024) W 1952 roku węgiersko-angielski matematyk Gabriel Andrew Dirac opracował twierdzenie mogące stwierdzić, czy graf jest Hamiltonowski. (Wilson, 2000) Dirac podał bezpośredni warunek, który głosił, że w przypadku, gdy w grafie prostym G każdy

z n wierzchołków, gdzie $n \geq 3$, graf jest Hamiltonowski w momencie, gdy dla każdego wierzchołka v zachodzi następujące równanie:

$$dev(v) \geq \frac{n}{2} \quad (13)$$

Gdzie:

- $\deg(v)$ wyraża stopień wierzchołka v w grafie;
- n – oznacza ilość wierzchołków w grafie;
- $\frac{n}{2}$ – oznacza połowę wierzchołków w grafie.

Warunek Diraca głosi, że w przypadku, gdy każdy wierzchołek w grafie prostym ma stopień co najmniej równy połowie wierzchołków w grafie, to taki graf posiada cykl Hamiltona. Twierdzenie to pozwala na szybką weryfikację potencjalnej hamiltonowskości grafu, bez konieczności przeszukiwania wszystkich możliwych ścieżek.

Natomiast według twierdzenia norweskiego matematyka Øysteina Orego z 1960 roku, w przypadku, kiedy graf prosty G ma n wierzchołków, gdzie $n \geq 3$ oraz dla każdej pary niesąsiadujących wierzchołków v i w suma stopni spełnia warunek opisany następującym wzorem:

$$\deg(v) + \deg(w) \geq n \quad (14)$$

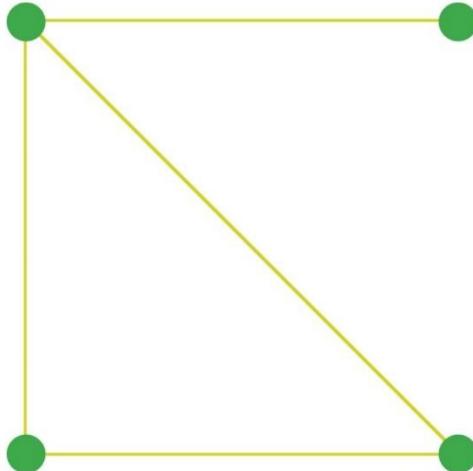
W równaniu (14) $\deg(v) + \deg(w)$ oznacza stopnie dowolnych niesąsiadujących ze sobą wierzchołków oznaczonych jako v oraz w . Symbol n stanowi ilość wierzchołków znajdujących się w danym grafie. W przypadku, gdy warunek zostanie spełniony i suma stopni każdej pary niepołączonych ze sobą krawędzią wierzchołków jest równa lub większa od ilości wierzchołków w grafie, to według twierdzenia Ore graf G jest hamiltonowski. Dowód Twierdzenia Ore oparty jest na dodaniu wszystkich krawędzi i analizie zmian w stopniach wierzchołków, doprowadzając tym samym do konstrukcji cyklu Hamiltona. Matematycznie cykl Hamiltona w grafie G można przedstawić jako ciąg wierzchołków:

$$v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_n \rightarrow v_1 \quad (15)$$

W ciągu przedstawionym w równaniu (15), każdy z wierzchołków ciągu odwiedzany jest wyłącznie jeden raz, natomiast sam cykl kończy się w tym samym punkcie, w którym rozpoczął, co spełnia zamkniętość cyklu.

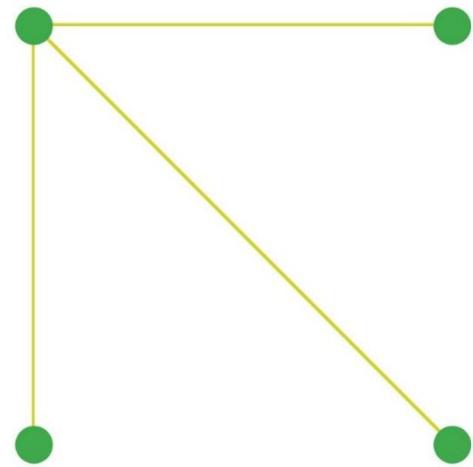
W sytuacji, gdy graf nie zawiera cyklu Hamiltona, jednak istnieje w nim ścieżka przechodząca przez każdy wierzchołek wyłącznie raz, jednak nie tworzy ona zamkniętego

cyklu to taki graf nazwano grafem półhamiltonowskim. Przykład takiego grafu przedstawiony został na rysunku 19.



Rysunek 19. Graf półhamiltonowski
[źródło: opracowanie własne na podstawie (Wilson, 2000)]

Z kolei grafem niehamiltonowskim jest graf, w którym nie istnieje cykl obejmujący każdy z wierzchołków wyłącznie jeden raz. Przykład takiego grafu został przedstawiony na poniższym rysunku 20.



Rysunek 20. Graf niehamiltonowski
[źródło: opracowanie własne na podstawie (Wilson, 2000)]

1.4.7. Problem cyklu Hamiltona

(Cormen, Leiserson, Rivest i Stein, Wprowadzenie do algorytmów, 2018) Problem cyklu Hamiltona poleca na odnalezieniu ścieżki grafu nieskierowanego, odwiedzającej każdy wierzchołek wyłącznie jeden raz, zamykając tym samym cykl poprzez powrót do wierzchołka początkowego. Problem można zdefiniować jako język formalny HAM-CYCLE= {G: G jest grafem hamiltonowskim}. Jest to zbiór wszystkich grafów G, które posiadają cykl Hamiltona. HAM-CYCLE określa nazwę rozwiązywanego problemu. {G} symbolizuje zbiór wszystkich grafów. Zapis G jest grafem hamiltonowskim stanowi kryterium, które muszą spełniać grafy należące do zbioru HAM-CYCLE, innymi słowy każdy graf G musi posiadać cykl Hamiltona.

Problem cyklu Hamiltona zaliczany jest do problemów NP-zupełnych. Innymi słowy każdy problem NP można sprowadzić do problemu cyklu Hamiltona w czasie wielomianowym, przez co jest on jednym z najtrudniejszych w teorii obliczeń. Dla instancji problemu HAM-CYCLE, algorytm weryfikacji wymaga dokonania sprawdzenia permutacji wierzchołków G oraz przeprowadzenia badania, czy któraś z nich stanowi cykl Hamiltona. Czas działania algorytmu weryfikacji jest ekspotencjalny względem liczby wierzchołków grafu, z powodu przeszukiwania $O(n!)$ permutacji.

Istnieje wiele technik rozwiązywania problemu cyklu Hamiltona m.in. poprzez użycie technik redukcji problemu NP-zupełnego. Przykładem redukcji problemu jest VERTEX-COVER. (Wikipedia, 2024) Pokrycie wierzchołkowe grafu stanowi zbiór wierzchołków wybranych tak, aby każda z krawędzi była incydentalna z przynajmniej jednym z nich. Znajdowanie minimalnego pokrycia wierzchołkowego stanowi typowy NP-trudny problem optymalizacyjny, co sugeruje, że nie istnieje algorytm wielomianowy, który byłby w stanie go rozwiązać, w przypadku, gdy $P \neq NP$. Problem ten jest również trudny do aproksymacji, ale istnieją proste algorytmy aproksymacyjne, które osiągają współczynnik 2.

(Cormen, Leiserson, Rivest i Stein, Wprowadzenie do algorytmów, 2018) Definicja problemu VERTEX-COVER głosi, że dany jest graf $G = (V, E)$, a także liczba k wierzchołków, które należy dobrać w taki sposób, aby każda z krawędzi grafu była incydentalna z przynajmniej jednym wierzchołkiem. Aby dokonać redukcji VERTEX-COVER do HAM-CYCLE należy skonstruować nowy graf G' powstały z grafu G . Graf G' posiada cykl Hamiltona wtedy i tylko wtedy, gdy graf G posiada pokrycie wierzchołkowe o rozmiarze k , bądź mniejszym. Na początku dla każdego z wierzchołków v grafu G należy skonstruować gadżet w G' . Gadżet będzie składał się z nowych wierzchołków oraz krawędzi, których celem jest wymuszenie przejścia cyklu wyłącznie przez określone wierzchołki, symulując tym samym

wybór wierzchołka v w pokryciu. Następnie między gadżetami w grafie G' należy umieścić krawędzie, które będą odpowiadały krawędzią w G . W przypadku wybrania wierzchołka w pokryciu w grafie G , krawędzie łącząc gadżety w G' zostaną połączone w taki sposób, aby możliwe było utworzenie cyklu Hamiltona. Można to wyrazić wzorem $VERTEX - COVER \leq_p HAM - CYCLE$. Oznacza to, że problem pokrycia wierzchołkowego jest redukowalny w sensie wielomianowym do problemu cyklu Hamiltona. Metoda ta jest kluczowa w badaniach nad NP-zupełnością.

2. Cel i zakres pracy

Celem pracy było przeprowadzenie implementacji oraz analizy efektywności wyselekcjonowanych algorytmów rozwiązujących problem komiwojażera (TSP - Travelling Salesman Problem). W ramach realizacji założonych celów dokonano porównania wybranych algorytmów pod kątem złożoności obliczeniowej oraz skuteczności w znajdowaniu optymalnych rozwiązań. Przeanalizowano dziewięć różnych algorytmów: algorytm dwuoptymalny, trójoptymalny, Kruskala, Prima, selekcji krawędzi, genetyczny, mrówkowy, Christofidesa oraz Little'a. Na podstawie przeprowadzonej analizy zdecydowano się na implementację następujących algorytmów:

- selekcji krawędzi;
- dwuoptymalnego;
- trój-optymalnego;
- Christofidesa.

Zakres pracy objął implementację wybranych algorytmów tj. dwuoptymalnego, trój-optymalnego, Christofidesa oraz selekcji krawędzi w językach C++, Python oraz z wykorzystaniem programu Matlab. Zaimplementowano również uniwersalny generator grafów losowych w języku C++. Wygenerowano dane wejściowe dla grafów o liczbie wierzchołków wynoszącej 20, 50, 100, 300, 500 oraz 1000. Badania przeprowadzono na trzech różnych środowiskach komputerowych. Wyznaczono czas uzyskania najlepszego rozwiązania oraz najkrótszą trasę dla każdego zestawu danych z osobna. Przeanalizowano również wyniki pod kątem efektywności algorytmów. Dokonano oceny złożoności obliczeniowej każdego z opracowanych algorytmów. Wyniki uzyskane z przeprowadzonych badań zostały przedstawione w tabelach, oddzielnie dla każdego języka programowania oraz każdego komputera, a także zilustrowane za pomocą wykresów liniowych oraz wykresu słupkowego.

3. Metodyka badań

W celu opracowania metodyki badań odpowiedniej dla niniejszej pracy magisterskiej przeprowadzono wszechstronny przegląd dostępnej literatury. Analizowano publikacje naukowe, książki, artykuły branżowe oraz źródła internetowe dotyczące zarówno teorii, jak i praktycznych zastosowań algorytmów rozwiązywania problemu komiwojażera. Kluczowe pozycje literatury, takie jak "Wprowadzenie do algorytmów" autorstwa Thomasa H. Cormena, Charlesa E. Leisersona, Ronalda L. Rivesta i Clifforda Steina, "Algorytmy optymalizacji dyskretnej z programami w języku Pascal" Sysły, Deo i Kowalika, „Wprowadzenie do teorii grafów” Wilsona oraz „Heurystyczny algorytm komiwojażera oparty na selekcji krawędzi” autorstwa Józefa Zielińskiego stanowiły podstawę teoretyczną do wyboru i analizy odpowiednich metod badawczych.

Na podstawie przeglądu literatury zdecydowano się na implementację i porównanie czterech algorytmów: selekcji krawędzi, Christofidesa, dwu-optymalnego oraz trój-optymalnego. Algorytmy te zaimplementowano w trzech językach programowania: C++, Python oraz Matlab, co pozwoliło na wszechstronną analizę ich efektywności. Wygenerowano grafy o różnej liczbie wierzchołków (20, 50, 100, 300, 500, 1000) za pomocą opracowanego generatora grafów, które stanowiły bazę do testowania wybranych algorytmów. Testy przeprowadzono na trzech różnych komputerach o zróżnicowanych specyfikacjach sprzętowych, co pozwoliło na ocenę wydajności algorytmów w różnych środowiskach sprzętowych, umożliwiając uzyskanie bardziej wiarygodnych wyników.

Dane dotyczące czasu wykonania algorytmów, ich złożoności obliczeniowej oraz jakości uzyskanych rozwiązań były zbierane i analizowane. Wyniki te były następnie porównywane w celu oceny efektywności poszczególnych algorytmów. Wybór algorytmów do badania był motywowany ich różnorodnością pod względem złożoności obliczeniowej i podejścia do rozwiązywania problemu komiwojażera. Algorytmy takie jak dwu-optymalny oraz trój-optymalny charakteryzują się prostotą i efektywnością w znajdowaniu przybliżonych rozwiązań, natomiast algorytm Christofidesa oferuje gwarancję jakości rozwiązania w określonych przypadkach. Algorytm selekcji krawędzi stanowił interesującą alternatywę, pozwalając na analizę jego skuteczności w porównaniu z bardziej znymi metodami.

Badania przeprowadzono w trzech różnych środowiskach komputerowych, co umożliwiło analizę wpływu specyfikacji sprzętowej na wyniki. Każdy z komputerów posiadał różne parametry techniczne, takie jak procesor, ilość pamięci RAM oraz system operacyjny, co pozwoliło na dokładne zrozumienie, jak te czynniki wpływają na wydajność algorytmów.

Wszystkie uzyskane dane były przetwarzane za pomocą narzędzi statystycznych oraz programistycznych w celu dokładnej analizy. Wyniki były prezentowane w formie wykresów i tabel, co umożliwiało łatwe porównanie efektywności poszczególnych algorytmów.

Metodyka badań została opracowana w taki sposób, aby mogła zostać powtórzona przez dowolną osobę zajmującą się podobną problematyką, co zapewnia transparentność i wiarygodność przeprowadzonych badań.

4. Przegląd algorytmów

Rozdział czwarty skupia swoją uwagę na przedstawieniu najpopularniejszych algorytmów do rozwiązywania problemu komiwojażera. Wśród opisanych algorytmów znajdują się algorytmy k-optymalne, zachłanne, a także algorytm selekcji krawędzi, genetyczny, mrówkowy, Christofidesa oraz Little'a.

4.1. Algorytmy k-optymalne.

(Anholcer, 2023) Algorytmy k-optymalne stanowią jedną z powszechnie znanych metod heurystycznych, służących do rozwiązywania problemu komiwojażera. Zadaniem wspomnianych heurystyk jest przeszukiwanie lokalne rozwiązania za pomocą iteracyjnych wymian k tras powstały wcześniejszym rozwiązaniem, podejmując próbę odnalezienia rozwiązania o niższym koszcie. W przypadku tych algorytmów należy rozważyć minimum $n = 2k$ miast, ze względu na fakt, iż tylko w takim przypadku istnieje możliwość odnaleźć k rozłącznych tras. Kiedy liczba miast n jest równa k, wtedy istnieje możliwość odnalezienia dokładnego rozwiązania. (Encyklopedia Algorytmów, 2017) Procedura algorytmu składa się z trzech kroków: usunięcia krawędzi, sprawdzenia połączeń oraz zapamiętania najlepszego rozwiązania. Na początek dokonuje się usunięcia krawędzi z cyklu k oraz zastąpienia ich innymi wybranymi krawędziami, w celu utworzenia prawidłowego cyklu. Usuwając krawędzie dokonywany jest podział cyklu na ilość k fragmentów. (Anholcer, 2023) Przykładowo dla cyklu:

$$C = (M_1, \langle i_1, j_1 \rangle, M_2, \langle i_2, j_2 \rangle) \quad (16)$$

gdzie M_1 oraz M_2 symbolizują fragmenty cyklu, natomiast $\langle i_1, j_1 \rangle$ oraz $\langle i_2, j_2 \rangle$ to krawędzie, które zostaną usunięte. Kolejny krok to dokonanie analizy każdego możliwego połączenia powstały wcześniejszych fragmentów oraz sprawdzenie czy taki wariant będzie lepszy od aktualnego. Dla przedstawionego wcześniejszej matematycznie przykładu istnieją dwie możliwości przekształcenia cyklu:

$$C' = (M_1, \langle i_1, i_2 \rangle, m_2, \langle j_2, j_1 \rangle) \quad (17)$$

$$C'' = (m_1, \langle j_1, j_2 \rangle, M_2, \langle i_2, i_1 \rangle) \quad (18)$$

W równaniu (17) m_2 symbolizuje odwrócony fragment M_2 , natomiast w równaniu (18) m_1 to odwrotność fragmentu M_1 . W przypadku, gdy nowe rozwiązanie jest optymalniejsze od odnalezione wcześniejszej, należy dokonać zapisu oraz kontynuować proces. (Encyklopedia

Algorytmów, 2017) Złożoność algorytmu określa się za pomocą liczby możliwych wariantów przekształceń cyklu co wyrażone jest wzorem:

$$\frac{n! \cdot 2^{k-1}}{k \cdot (n-k)!} \quad (19)$$

Gdzie:

- k – symbolizuje poziom optymalności
- n – ilość krawędzi

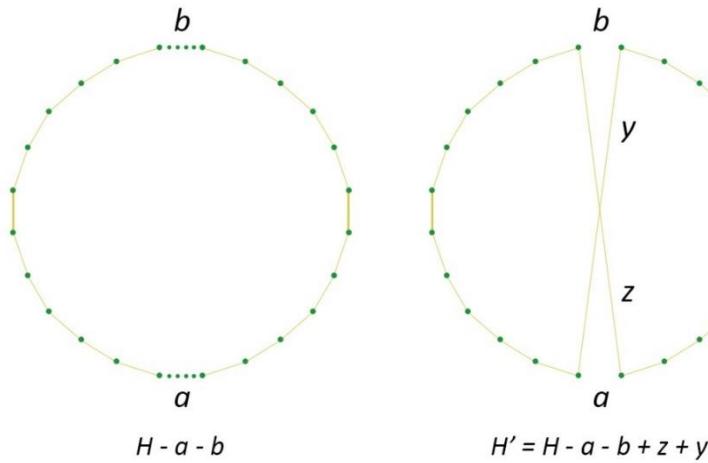
Złożoność rzędu $O(n^k)$ nawet w przypadku dużych złożoności k nie przekroczy $n!$.

Najpopularniejsze heurystyki k-optymalne to algorytm dwu-optymalny oraz trój-optymalny.

4.1.1. Algorytm dwu-optymalny

(Sidford, 2020) Algorytm dwu-optymalny jest jedną z popularnych heurystyk, które zostały zastosowane w problemie komiwojażera. Heurystyka ta polega na iteracyjnej poprawie danego rozwiązania przez lokalne zmiany. W algorytmie tym cykl obejmujący każdy wierzchołek grafu, podlega stopniowej optymalizacji za pomocą zmiany dwóch nieprzylegających krawędzi, powodując powstanie nowego cyklu. Proces powtarzany jest do momentu, w którym nie ma możliwości uzyskania krótszej ścieżki poprzez dokonywania kolejnych zmian. (Sysło, Deo i Kowalik, 1995) Na początku należy wybrać losowy bądź heurystycznie ustalony początek cyklu H oraz dokonać usunięcia dwóch łuków. Kolejnym krokiem jest utworzeniu cyklu H' zastępując usunięte krawędzie innymi. W przypadku, gdy koszt cyklu H' będzie mniejszy od cyklu H , należy zastąpić cykl pierwotny zastąpić cyklem H' oraz kontynuować zamianę łuków. Jednak w przypadku, gdy koszt cyklu H jest mniejszy, należy dokonać w nim zamiany innych krawędzi. Proces ten powtarzany jest do momentu uzyskania najlepszej trasy. W trakcie każdorazowego przebiegu algorytmu przeprowadza się badanie $\frac{n \cdot (n-3)}{2}$ par łuków, natomiast złożoność czasowa pojedynczej iteracji wynosi $O(n^2)$.

Proces wymiany dwu-optymalnej przedstawiony został na rysunku 21.



Rysunek 21. Wymiana 2-optimalna
[źródło: opracowanie własne na podstawie (Sysło, Deo i Kowalik, 1995)]

Matematyczne uzasadnienie wymiany 2-optimalnej opiera się na porównaniu wag tras przed i po zaproponowanej zmianie. W przypadku, gdy różnica w wagach, która określana jest jako $\delta = w(H) - w(H')$ daje wynik dodatni, wtedy nowo utworzona trasa H' określana jest jako krótsza od H . Przypadek ten jest istotny, aby kryterium zamiany było większe od zera, co sugeruje możliwość skrócenia trasy. Kryterium zamiany opisane jest wzorem:

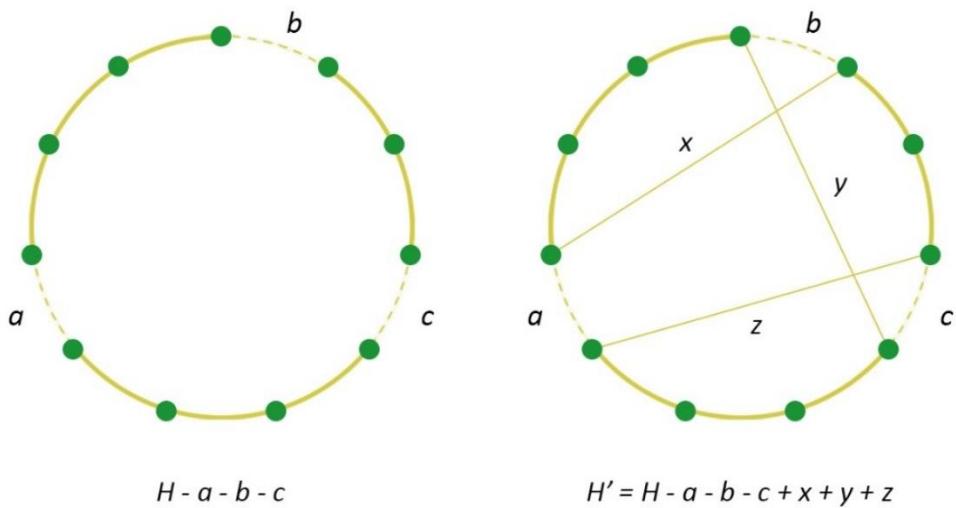
$$w(x_i) + w(x_j) - (w(y_p) + w(y_q)) \quad (20)$$

Kryterium zamiany przedstawione w równaniu 20 zawiera wagi krawędzi przed oraz po dokonaniu zamiany. Pierwsza część równania $w(x_i) + w(x_j)$ przedstawia wagi oryginalnych krawędzi w trasie, które rozważane są do zmiany. Oznaczają one bezpośrednie połączenie pomiędzy wierzchołkami, które będą zamieniane. Droga część równania $w(y_p) + w(y_q)$ stanowi wagi krawędzi otrzymane po zamianie. y_p oraz y_q reprezentują nowe połączenia powstałe po zamianie krawędzi (x_i, x_{i+1}) oraz (x_j, x_{j+1}) .

4.1.2. Algorytm trój-optymalny

(Sysło, Deo i Kowalik, 1995) Algorytm trój-optymalny to heurystyka wykorzystywana do rozwiązywania problemu komiwojażera, polegająca na iteracyjnej poprawie istniejącej trasy poprzez zamianę trzech krawędzi. Operacja ma na celu skrócenie całkowitej długości trasy, jednocześnie sprawdzając różne opcje ponownego połączenia fragmentów trasy po usunięciu wybranych krawędzi. Na początku każdego kroku algorytm wybiera trzy krawędzie, które zostaną usunięte, a następnie ponownie łączy powstałe trzy segmenty w innej kolejności, analizując w ten sposób czy istnieje możliwość osiągnięcia krótszej trasy, tworząc tym samym

nowe cykle. Każda z tras, które zostały utworzone jest oceniana pod kątem jej długości. W przypadku, gdy nowy cykl jest krótszy od poprzedniego, wtedy zmiana jest akceptowana. Operacja powtarzana jest do momentu, kiedy nie ma możliwości kontynuowania popraw bądź osiągnięta zostanie określona liczba iteracji. Algorytm trój-optymalny ma potencjał do odnajdywania zdecydowanie krótszych cykli, niżeli algorytm dwu-optymalny. Dodatkowym atutem tej metody jest większa złożoność, przez co może lepiej eksplorować przestrzeń rozwiązań. Mimo, iż algorytm ten jest bardziej skuteczny w wyszukiwaniu krótszych cykli, nie zapewnia on odnalezienia zawsze najkrótszego możliwego cyklu. Dodatkowym ograniczeniem jest jego czasochłonność, ze względu na znacznie większą ilość kombinacji do sprawdzenia przy każdej z iteracji. Rysunek 22 przedstawia przykładową wymianę trój-optymalną.



Rysunek 22. Wymiana trój-optymalna
[źródło: opracowanie własne na podstawie (Sysło, Deo i Kowalik, 1995)]

4.2. Algorytm zachłanny

(CodingDrills, 2024) Algorytm zachłanny to jedna z metod optymalizacji, której celem jest podejmowanie lokalnych optymalnych decyzji, w każdym kroku działania algorytmu, dążąc do globalnie optymalnego rozwiązania. W kontekście problemu komiwojażera, celem algorytmu zachłannego jest wyszukanie najkrótszej możliwej drogi, za pomocą której będzie możliwość odwiedzenia wszystkich możliwych miast wyłącznie jeden raz, powracając na samym końcu do miejsca rozpoczęcia drogi. Na początku procesu należy dokonać wyboru miasta startowego $C \in V$, gdzie V określa zbiór wszystkich miast. Następnie należy dodać miasto C do zbioru odwiedzonych miast $S: S = C$. Tak długo jak wszystkie miasta nie zostaną odwiedzone $S \neq V$, należy podjąć próbę odnalezienia najbliższego nieodwiedzonego miasta,

oznaczonego symbolem j , minimalizując jednocześnie odległość $d(C, j)$, gdzie d symbolizuje funkcję odległości oraz wyraża się wzorem:

$$j = \arg \min_{v \in V \setminus S} d(C, v) \quad (21)$$

Gdzie:

- $\arg \min$ – symbolizuje operator matematyczny, zwracający wartość zmiennej, dla której dana funkcja osiąga minimum;
- $v \in V \setminus S$ – oznacza, że przeszukiwany jest zbiór miast nie odwiedzonych;
- V - zbiór wszystkich miast;
- S – zbiór miast odwiedzonych ;
- $V \setminus S$ – wyraża różnicę zbiorów V oraz S , wszystkie miasta z V , których nie ma w zbiorze S ;
- $d(C, v)$ – wyraża funkcję odległości między miastem bieżącym C , a miastem v .

Kolejny krok to dodanie miasta j do zbioru odwiedzonych miast $S: S = S \cup j$, oraz ustawienia miasta C , jako miasto $j | C = j$. Na zakończenie należy dodać krawędź powrotną do miasta, w którym droga została rozpoczęta, w celu zamknięcia cyklu i zakończenia trasy. Wyrażone zostało to wzorem $S = S \cup C$. Przedstawione matematyczne sformułowania najprościej przedstawić na prostym przykładzie, w którym rozważona została macierz odległości dla czterech miast A, B, C oraz D. Rysunek 23 przedstawia macierz odległości między miastami przykładowego zadania.

$$d = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{bmatrix}$$

Rysunek 23. Macierz odległości pomiędzy miastami
[źródło: opracowanie własne]

Na początku zgodnie z zasadą algorytm zachłanny rozpocznie od miasta A, wybierając najbliższe nieodwiedzone miasto, aż do momentu odwiedzenia każdego z miast. Należy ustawić zbiór miast odwiedzonych $S = \{A\}$ oraz wskazać bieżące miasto $C = A$. W kolejnym kroku należy odnaleźć najbliższe nieodwiedzone miasto j , minimalizując odległość $d(C, j)$. Rozważane zostają odległości pomiędzy miastami $Bd(A, B) = 10$, $Cd(A, C) = 15$ oraz

$Dd(A, D) = 20$ względem miasta początkowego A. Według przeprowadzonej analizy wybrane ze względu na najmniejszą odległość do miasta A zostaje miasto B. Następnie należy dokonać aktualizacji $C = B$ oraz dodanie miasta B do zbioru odwiedzonych miast $S = A, B$. Ponownie następuje wyszukanie najbliższego nieodwiedzonego miasta względem miasta, tym razem względem punktu B. Pod uwagę brane są miasta $Cd(B, C) = 35$ oraz $Dd(B, D) = 25$. Wybór ze względu na mniejszą odległość pada na miasto D, dokonywana jest aktualizacja $C = D$ oraz dodanie miasta D do zbioru miast odwiedzonych $S = A, B, D$. Operacja wyszukania najbliższego nieodwiedzonego miasta jest ponawiana, jednak ze względu, że pozostało tylko miasto C, dlatego względem niego odliczana jest odległość $d(D, C) = 30$. Dokonywana jest aktualizacja obecnego miasta $C = C$ oraz następuje dodanie ostatniego miasta do zbioru miast odwiedzonych $S = A, B, C, D$. Następnie dodawana jest krawędź powrotna do miasta startowego A. Ostateczna trasa prezentuje się w następujący sposób: $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$, a jej długość całkowita wynosi $10 + 25 + 30 + 15 = 80$.

Algorytmy zachłanne nie zawsze gwarantują uzyskanie optymalnego rozwiązania, mimo to dla niektórych są one wystarczające. Do algorytmów dokładnych zalicza się m.in. Algorytm Prima, Algorytm Kruskala, natomiast do niedokładnych algorytm najmniejszej krawędzi oraz najbliższego sąsiada.

4.2.1. Algorytm Kruskala

(Cormen, Leiserson, Rivest i Stein, Wprowadzenie do algorytmów, 2007) Algorytm Kruskala reprezentuje algorytmy służące do znajdowania minimalnego drzewa rozpinającego MST⁷ w grafie ważonym. (Wojciechowski i Pieńkocz, 2013) Został opracowany i zaprezentowany przez Josepha Bernarda Kruskala w 1956 roku. Jego działanie opiera się na wyborze krawędzi o najmniejszej wagie, które nie tworzą cyklu z wybranymi wcześniej krawędziami, natomiast suma wag krawędzi jest minimalna. (Sysło, Deo i Kowalik, 1995) Jeżeli wystąpi sytuacja, w której poddana badaniu krawędź osiągnie cykl z wybranymi wcześniej krawędziami, wtedy krawędź ta jest pomijana. Proces trwa, dopóki nie zostanie wybrane n-1 krawędzi, bądź wszystkie krawędzie w grafie nie zostaną rozpatrzone. Formalnie dla grafu $G = (V, E)$, gdzie V to zbiór wierzchołków, natomiast E to zbiór krawędzi proces ten można opisać w następujący sposób:

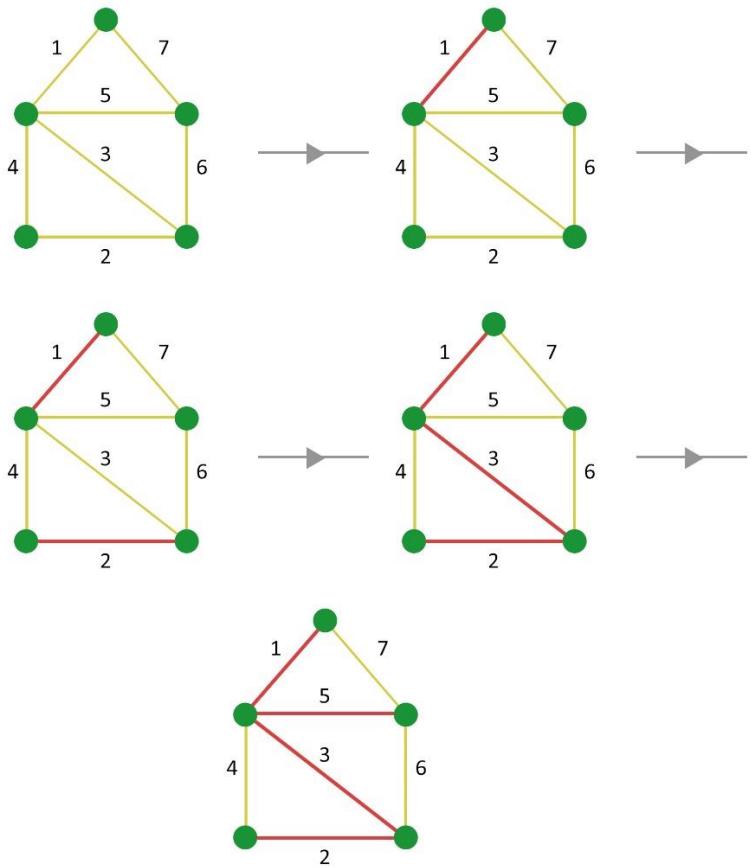
⁷ MST – Minimum Spanning Tree – minimalne drzewo rozpinające grafu ważonego to podzbiór krawędzi, który łączy wszystkie wierzchołki grafu w taki sposób, że suma wag tych krawędzi jest minimalna i nie zawiera cykli.

- Inicjalizacja – Należy utworzyć zbiór A, który początkowo będzie zbiorem pustym $A = \emptyset$. Następnie trzeba utworzyć zbiór, w którym każdy wierzchołek jest reprezentowany przez osobne drzewo $\forall v \in V, T_v = v$;
- Sortowanie krawędzi – Należy posortować krawędzie w zbiorze E według wag $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$;
- Dodawanie krawędzi – W tym kroku należy przejść przez posortowane krawędzie dodając każdą z nich do zbioru A w przypadku, gdy nie tworzą one cyklu z innymi wybranymi krawędziami. W przypadku, gdy wierzchołki u_i oraz v_i ze zbioru $e_i = (u_i, v_i)$ w zbiorze krawędzi E, należą do różnych drzew $FIND - SET(u_i) \neq FIND - SET(v_i)$, należy dodać e_i do zbioru A: $A \leftarrow A \cup e_i$ oraz połączyć drzewa zawierające wierzchołki u_i oraz v_i za pomocą funkcji do zarządzania drzewami $UNION(T_{u_i}, T_{v_i})$;
- Zakończenie – W przypadku osiągnięcia w zbiorze A $n - 1$ krawędzi, należy zwrócić zbiór A jako minimalne drzewo rozpinające.

Czas działania algorytmu Kruskala w grafie $G = (V, E)$ zależy od operacji sortowania, jak i operacji na zbiorach rozłącznych. Zakładając, że zastosowane zostały zastosowane struktury danych z heurystyką łączenia zarówno według rangi, jak i kompresji ścieżek, czas działania algorytmu wynosi $O(E \log E)$ dla sortowania krawędzi oraz $O(E\alpha(V))$ dla operacji FIND – SET⁸ oraz UNION⁹ wykonywanych w pętli, gdzie $\alpha(V)$ stanowi funkcję odwrotną do funkcji Ackermanna, która charakteryzuje się bardzo powolnym wzrostem oraz możliwością traktowania jej jako stałej w praktycznych zastosowaniach. Złożoność czasowa wynosi $O(E \log E)$, bądź jest równoważne $O(E\alpha(V))$, w przypadku, gdy E jest maksymalnie równe V^2 .

⁸ FIND-SET - Operacja w strukturze zbiorów rozłącznych, która zwraca reprezentanta zbioru zawierającego dany element.

⁹ UNION - Operacja w strukturze zbiorów rozłącznych, która łączy dwa zbiory w jeden.

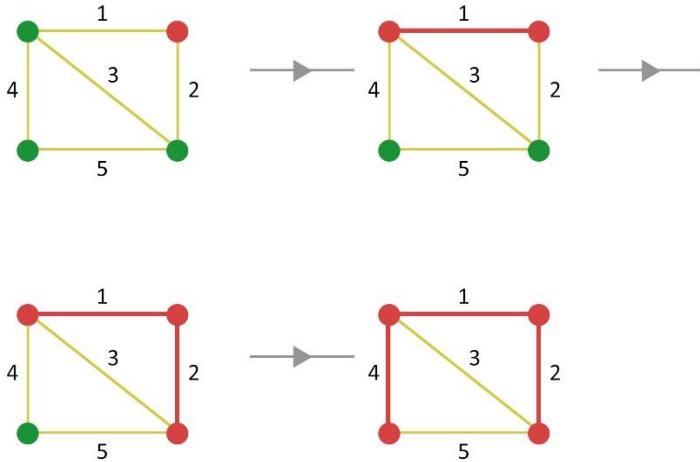


Rysunek 24. Proces działania algorytmu Kruskala
[źródło: opracowanie własne na podstawie (Yadav, 2022)]

Proces działania algorytmu Kruskala warto przedstawić na prostym przykładzie. (Yadav, 2022) W przedstawionym powyżej przykładzie zawartym na rysunku 24 na początku wybrana została krawędź o najmniejszej wadze, wynoszącej 1. Krok ten jest powtarzany dla kolejnej krawędzi posiadającej najmniejszą wagę, w tym przypadku jest to krawędź numer 2. Można zaobserwować, że wybrane krawędzie nie są ze sobą w żaden sposób połączone, więc w kolejnym etapie wybrana została krawędź o najmniejszym koszcie wynoszącym 3, która dodatkowo połączy poprzednio wybrane krawędzie ze sobą. Ze względu na fakt, że połączenie wybranych dotąd krawędzi z krawędziami o kosztach 4,6 oraz 8 spowodowałoby utworzenie cyklu, odcinki te zostają pominięte, a do zbioru krawędzi dobiera się krawędź o wadze równej 5. Po zsumowaniu wszystkich wag wybranych krawędzi otrzymany zostaje koszt drzewa rozpinającego, który wynosi $1 + 2 + 3 + 5 = 11$.

4.2.2. Algorytm Prima

(Cormen, Leiserson, Rivest i Stein, Wprowadzenie do algorytmów, 2018) Algorytm Prima to przedstawiciel algorytmów, których celem jest odnalezienie minimalnego drzewa rozpinającego w grafie ważonym. Po raz pierwszy algorytm ten został opublikowany w 1930 roku przez czeskiego matematyka Vojtěcha Jarníka, natomiast niezależnie odkryty został w 1957 roku przez amerykańskiego matematyka Roberta C. Prima III. (Wojciechowski i Pieńkocz, 2013) Działanie algorytmu opiera się na rozbudowaniu drzew, za pomocą dodania kolejnych krawędzi posiadających najmniejszy koszt, które łączą wierzchołki będące już w drzewie z wierzchołkami, które nie są obecnie dostępne w drzewie. Sam proces trwa do momentu włączania wszystkich wierzchołków do drzewa. Kroki algorytmu Prima można przedstawić w kilku prostych krokach. Na samym początku należy wskazać dowolny wierzchołek u z grafu $G = (V, E)$, który będzie pełnił funkcję wierzchołka startowego. Kolejny krok to stworzenie dwóch zbiorów: V_1 , który będzie zawierał wierzchołki drzewa, a także E_1 , pełniący funkcję zbioru zawierającego krawędzie drzewa. W tym momencie dokonywana zostaje inicjalizacja zbiorów jako $V_1 := u$ oraz $E_1 := \emptyset$, a następnie należy dokonać ustalenia początkowego kosztu drzewa jako $wt := 0$. Kolejny etap to iteracja, w której należy wybrać krawędź $e = u, v$ posiadającą minimalny koszt względem innych krawędzi, gdzie $u \in V_1$ oraz $v \notin V_1$. Następnie należy dodać wierzchołek v do zbioru V_1 , a także e do zbioru E_1 oraz dokonać aktualizacji kosztu drzewa $wt + w(e)$, gdzie $w(e)$ stanowi wagę krawędzi e . Podsumowując iterację $T = (V, E)$ określone jest jako drzewo minimalnie rozpinające, a jego koszt wynosi wt . Złożoność czasowa algorytmu zależy od użytej struktury danych. W przypadku zastosowania macierzy sąsiedztwa wynosi ona $O(n^2)$. Z kolei w wypadku zastosowania kopca binarnego złożoność czasowa wynosi $O(q \log n)$. Istnieje również możliwość zastosowania kopca Fibonacciego, w którym złożoność wynosi $O(q + n \log n)$. Proces działania algorytmu Prima został przedstawiony na rysunku 25.



Rysunek 25. Proces działania algorytmu Prima
[źródło: opracowanie własne na podstawie (Vasava, 2019)]

W przypadku algorytmu Prima analizę należy rozpocząć od dowolnie wybranego wierzchołka. Kolejny krok to wskazanie nowego wierzchołka, który sąsiaduje z wcześniej wybranym wierzchołkiem. (Vasava, 2019) W przykładzie przedstawionym powyżej na rysunku X na początku wybrana została krawędź, której waga wynosiła 1. Kolejny krok to wybór odpowiedniej krawędzi z pozostałych o wagach wynoszących 2, 3 oraz 4. Dokonano wyboru odcinka, którego koszt wynosi 2. Kontynuując iteracje do wyboru pozostają krawędzie 3,4 oraz 5. Z uwagi, że wybór krawędzi 3 spowodowałby utworzenie cyklu, wybrana zostaje krawędź z wagą 4. Reasumując koszt powstałego MST wynosi $1+2+4 = 7$.

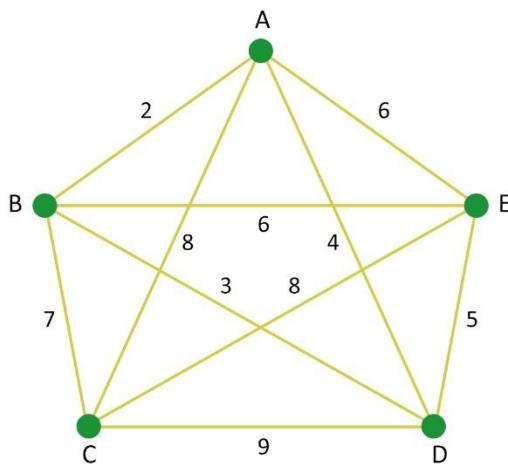
4.3. Algorytm selekcji krawędzi

(Zieliński, Heurystyczny algorytm komiwojażera oparty na selekcji krawędzi, 2024) Algorytm selekcji krawędzi w odróżnieniu od innych znanych algorytmów nie zajmuje się budową drzew oraz nie tworzy żadnych cykli pomocniczych. Utworzenie tego algorytmu zależne jest od faktu, czy spełnione zostaną następujące postulaty:

1. W grafie posiadającym cykl Hamiltona, stopnie wszystkich wierzchołków wynoszą 2;
2. W grafie musi istnieć przynajmniej jeden cykl Hamiltona;
3. Krawędzie cyklu Hamiltona posiadają najmniejsze możliwe wagi.

Algorytm w nieuporządkowany sposób wybiera krawędzie, które spełniają powyższej wymienione wymagania. Rozpoczyna pracę od stworzenia listy m krawędzi, następnie sortując

je według wag. Następnie graf pusty o n wierzchołkach przekształcany jest w graf regularny¹⁰ stopnia drugiego tj. krawędzie ze zbioru M zostają dołączone n krawędzie z zastosowaniem sposobu rosnących wag. Do dołączenia wybierane są wyłącznie krawędzie nie doprowadzające do zwiększenia jakiegokolwiek wierzchołka do większej wartości niż 2. Dodatkowo cykl nie powinien zostać przedwcześnie zamknięty z powodu dołączenia krawędzi, z wyjątkiem krawędzi ostatniej, której zadaniem jest zamknięcie cyklu Hamiltona. Na rysunku 26 przedstawiono przykładowy graf regularny.



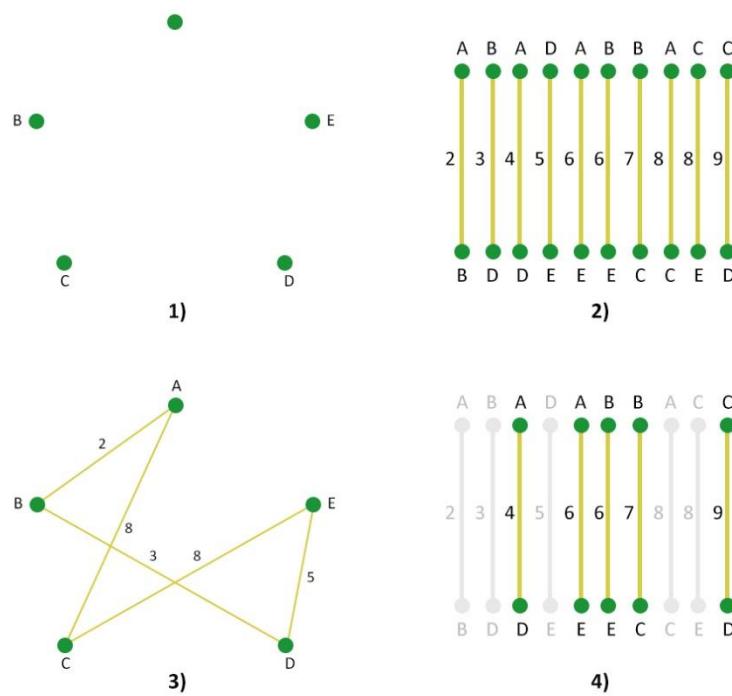
Rysunek 26. Przykładowy graf regularny
[źródło: opracowanie własne na podstawie (Wilson, 2000)]

Strukturę danych stanowi zbiór krawędzi, oznaczony symbolem E , a także zbiór wierzchołków oznaczony jako V . Krawędzie e muszą spełniać następujący warunek: pierwszy incydenty wierzchołek v_1 , drugi incydenty wierzchołek v_2 oraz posiadać wagę w . Z kolei wszystkie wierzchołki v spełniają warunek: sąsiadni wierzchołek to v_1 , sąsiadni wierzchołek to v_2 oraz stopień wierzchołka r . Ponadto istnieje licznik krawędzi, którego zadaniem jest zliczenie krawędzi uwzględnionych dotychczas w trasie komiwojażera.

Należy rozpocząć od wprowadzenia do zbioru E danych grafu oraz zresetowania licznika krawędzi oraz dodatkowo dokonania resetu w zbiorze wierzchołków V . Algorytm sortuje od najmniejszego elementu zbioru E oraz w odpowiedni sposób przenosi dane zgromadzone w zbiorze E , do wyzerowanego zbioru V . Jako pierwsza wprowadzana jest krawędź posiadająca najmniejszą wagę, w sposób, w którym w wierzchołkach v_1 oraz v_2 , które są połączone krawędzią ze zbioru V dodaje się stopień, a także numer sąsiadującego wierzchołka. W ten sam sposób wprowadzana jest również druga krawędź.

¹⁰ Graf regularny - graf, w którym każdy wierzchołek ma ten sam stopień.

W kolejnych przypadkach można wprowadzić krawędzie wyłącznie, gdy ich incydentalne wierzchołki posiadają stopień 0 bądź 1. W przypadku, gdy oba wierzchołki posiadają stopień 1, należy sprawdzić czy cykl nie powstanie za wcześnie. To czy cykl istnieje sprawdzane jest poprzez przechodzenie od wierzchołka v_1 do sąsiednich wierzchołków o stopniu równym 2 do samego końca, czyli wierzchołka, którego stopień jest równy 1. W przypadku, gdy skrajny wierzchołek badanej krawędzi to drugi wierzchołek incydentalny v_2 , krawędź badana nie może zostać zaliczona do rozwiązań, ze względu na wystąpienie cyklu. Zbiór rozwiązań poszerzany jest o krawędź, której wierzchołek różni się od v_2 . Algorytm kończy działanie, gdy licznik dołączonych krawędzi będzie taki sam, jak liczba wierzchołków w grafie.



Rysunek 27. Proces działania algorytmu selekcji krawędzi
[źródło: opracowanie własne na podstawie (Wilson, 2000)]

Na rysunku 27 przedstawiona została zasada działania algorytmu selekcji krawędzi podzieloną na cztery etapy. W etapie pierwszym zainicjalizowano algorytm przez wprowadzenie danych grafu do zbioru E i wyzerowanie wszystkich elementów składowych zbioru wierzchołków V oraz licznika krawędzi. Następnie dokonano wstępnego kroku algorytmu, polegającego na posortowaniu elementów zbioru E według wag, od najmniejszej do największej. Wprowadzenie krawędzi według tej kolejności miało na celu minimalizację sumarycznej wagi drzewa. Pierwszą, najmniejszą krawędź wprowadzono do zbioru V poprzez inkrementację stopnia w dwóch wierzchołkach v_1 oraz v_2 incydentnych tej krawędzi oraz

wprowadzenie numeru sąsiadującego wierzchołka. Podobnie wprowadzono drugą krawędź. Każdą kolejną krawędź można było wprowadzić tylko wtedy, gdy jej incydentne wierzchołki miały stopień 0 lub 1. Jeśli oba wierzchołki miały stopień 1, sprawdzono, czy nie powstanie „przedwczesne” cykl. Cykl powstawał jedynie przy wprowadzaniu ostatniej krawędzi. Sprawdzenie istnienia cyklu przy dołączaniu krawędzi odbywało się poprzez przechodzenie kolejnych sąsiadujących wierzchołków o stopniu 2 aż do końca, czyli do wierzchołka o stopniu 1. Jeśli końcowym wierzchołkiem był drugi incydentny wierzchołek v_2 badanej krawędzi, oznaczało to powstanie cyklu, więc badaną krawędź nie można było zaliczyć do rozwiązania. Jeśli końcowy wierzchołek był różny od v_2 , badaną krawędź dołączano do zbioru rozwiązania. Działanie algorytmu kończyło się, gdy licznik dołączonych krawędzi osiągał wartość równą liczbie wierzchołków grafu minus jeden. Rysunek ilustruje kolejne etapy przetwarzania grafu, od inicjalizacji i sortowania, przez budowanie drzewa, aż po sprawdzanie i dołączanie krawędzi z zachowaniem zasad minimalizacji i unikania przedwczesnego cyklu.

Algorytm dzięki swojej prostocie jest bardzo wydajny i daje bardzo dobre wyniki. Dla przykładu przedstawionego na rysunku 27 optymalne rozwiązanie wynosi 26. W przypadku bardziej złożonych grafów możliwe jest osiągnięcie w bardzo szybki sposób dobre wyniki, ponieważ algorytm zawiera duże rezerwy. Istnieje możliwość iteracji algorytmu przy zmienionych warunkach początkowych, co umożliwia przesunięcie przed posortowany ciąg krawędzi pierwszej krawędzi pominiętej w pierwszej iteracji.

4.4. Algorytm genetyczny

(Debudaj-Grabysz, Deorowicz i Widuch, 2012) Algorytmy genetyczne są to metaheurystyczne techniki optymalizacyjne, za pomocą których istnieje możliwość rozwiązywania złożonych problemów optymalizacyjnych, a także poszukiwanie globalnych ekstermów funkcji. Za ich pomocą istnieje możliwość symulacji procesów biologicznej ewolucji m.in. selekcji naturalnej, krzyżowania, czy też mutacji w celu przekształcania populacji ewentualnych rozwiązań ku lepszemu przystosowaniu do zadanej funkcji celu. Algorytmy genetyczne zostały przedstawione w 1975 roku przez amerykańskiego naukowca Johna Hollanda z Uniwersytetu Michigan. Do podstawowych elementów algorytmów genetycznych zalicza się:

- Reprezentacje rozwiązań;
- Generowanie populacji początkowej;
- Ocena jakości chromosomów;
- Selekcja;

- Selekcja proporcjonalna;
- Selekcja rankingowa;
- Selekcja turniejowa;
- Krzyżowanie;
- Mutacja;
- Warunek zatrzymania.

W etapie pierwszym chromosomy pełną funkcję przedstawicieli możliwych rozwiązań problemu. Każdy z nich składa się z genów, które mają możliwość przyjęcia wszelakich wartości tzw. alleli. Najczęściej chromosomy w algorytmach genetycznych kodowane są binarnie. Początkowo dochodzi do wygenerowania populacji w losowy sposób, przez co każdy chromosom x_i stanowi ciąg bajtów o długości l . Można to wyrazić za pomocą wzoru: $Populacja = \{x_1, x_2, \dots, x_n\}$. Następnie każdy z chromosomów oceniany jest za pomocą funkcji przystosowania, której zadaniem ustalenie na ile prawidłowo dany każdy z chromosomów rozwiązuje problem. Funkcje przystosowania można zapisać za pomocą wzoru:

$$f(x) = \sum_{i=1}^n q_i x_i - K \max \left\{ \sum_{i=1}^n w_i x_i - W, 0 \right\} \quad (22)$$

Gdzie:

- $\sum_{i=1}^n q_i x_i$ – całkowita korzyść q_i , związana z x_i ;
 - x_i – zmienna binarna wskazująca czy element został wybrany (1), czy nie (0);
 - q_i – wartość związana z i ;
 - $K \max \{\sum_{i=1}^n w_i x_i - W, 0\}$ – kara za przekroczenie ograniczeń wagowych
 - K – współczynnik wagowy;
 - $\max \{\sum_{i=1}^n w_i x_i - W, 0\}$ – całkowita waga przekraczająca dopuszczalny limit W .
- Jeżeli waga całkowita jest mniejsza lub równa W , wartość wyrażenia jest równa 0.

Współczynnik wagowy wyznaczany jest za pomocą wzoru:

$$K = \frac{\max_{i=1,\dots,n} q_i}{\min_{i=1,\dots,n} w_i} \quad (23)$$

Gdzie:

- $\max_{i=1,\dots,n} q_i$ – symbolizuje maksymalną wartość korzyści spośród elementów
- $\min_{i=1,\dots,n} w_i$ – oznacza minimalną wagę spośród wszystkich elementów

W kolejnym etapie dokonuje się selekcji chromosomów do reprodukcji oraz sukcesji. Najczęściej stosuje się selekcje proporcjonalną opartą na metodzie ruletki, w której każdy z osobników x obecnej populacji przyporządkowuje się prawdopodobieństwo reprodukcji p_x , którą można opisać wzorem:

$$p_x = \frac{f_x}{\sum_{j=1}^N f_j} \quad (24)$$

We wzorze (24) f_x symbolizuje funkcję oceny chromosomu, a zatem szansa na wylosowanie chromosomu jest wprost proporcjonalna do wartości funkcji przystosowania. Aby wskazać pulę rodzicielską chromosomów, koło ruletki, której obwód wynosi 1 należy podzielić je na wycinki o długości łuku p_x . Kolejny krok to wylosowanie liczby z przedziału $[0,1)$ wskazującą punkt na obwodzie określonego wycinka koła ruletki. W przypadku funkcji przystosowania wyrażonej wzorem (24) istnieje prawdopodobieństwo wystąpienia ujemnych wartości. Aby tego uniknąć należy zastosować skalowanie przystosowania, które oblicza się według wzoru:

$$p_x = \frac{f_x - f_{min}}{\sum_{j=1}^N f_j - f_{min}} \quad (25)$$

We wzorze (25) f_{min} symbolizuje funkcję przystosowania najgorszego osobnika.

Istnieje również drugi wariant dokonania selekcji – selekcja rankingowa, w której chromosomy posortowane są wedle nierosnących wartości funkcji oraz nadaje im się numery rozpoczynające się od 0. Najczęściej stosowana jest funkcja liniowa określona wzorem:

$$p_x = a + k \left(1 - \frac{r_x}{r_{max}} \right) \quad (26)$$

We wzorze (26) symbole a oraz k są parametrami, przy których wyborze należy uwzględnić następujące warunki:

$$\sum_{i=1}^n p_x = 1, \quad 0 \leq p_x \leq 1 \quad (27)$$

Warto również określić, że jeżeli $r_x \geq r_y$, to $p_x \leq p_y$. Metoda selekcji rankingowa jest odpowiednia w przypadku zarówno problemów minimalizacji, jak i maksymalizacji.

Selekcja turniejowa polega na losowym wyborze kilku osobników z populacji, którzy rywalizują ze sobą w "turnieju". Zwycięzca turnieju, czyli osobnik o najwyższej wartości funkcji przystosowania, jest wybierany do dalszego przetwarzania, na przykład

do krzyżowania. Proces selekcji turniejowej rozpoczyna się od losowego wyboru k osobników z bieżącej populacji. Losowanie może zostać przeprowadzone zarówno z powtórzeniami, jak i bez. Wybrani osobnicy tworzą grupę turniejową, z której zostanie wyłoniony zwycięzca. Każdy z chromosomów oceniany jest za pomocą funkcji przystosowania $f(x)$, której zadaniem jest zmierzenie jakości rozwiązania jakie dany osobnik reprezentuje. Zwycięzcą jest osobnik x_{best} , który ma najwyższą wartość funkcji przystosowania spośród wszystkich uczestników turnieju. Sam wybór zwycięzcy można opisać wzorem:

$$x_{best} = \operatorname{argmax}_{x \in T} f(x) \quad (28)$$

Następnie rozważana jest opcja przeprowadzenia wymiany pokoleń. Najpopularniejszą metodą sukcesji jest całkowite zastępowanie, w której populacja potomna staje się bieżącą. Innym sposobem jest sukcesja z częściowym zastępowaniem, w której część najlepszych chromosomów rodzicielskich przekazywana jest do nowego pokolenia bez jakichkolwiek zmian, a tym samym nie podlegają operacji krzyżowania oraz mutacji. Na początku dokonuje się usunięcia $\eta \times n$ osobników populacji bazowej, a następnie w ich miejsce wprowadza się osobniki potomne. η symbolizuje współczynnik wymiany. Dokonuje się usunięcia chromosomów podobnych do potomnych bądź najgorzej przystosowanych. W przypadku sukcesji elitarnej dokonuje się skopiowania przynajmniej najlepszego osobnika z poprzedniego pokolenia na samym początku nowej generacji.

Kolejnym etapem jest krzyżowanie, które poleca na wymianie fragmentów chromosomów pomiędzy dwoma rodzicami w celu stworzenia potomstwa. Istnieją różne metody krzyżowania, takie jak krzyżowanie jednopunktowe, dwupunktowe i krzyżowanie z częściowym odwzorowaniem (PMX). Krzyżowanie jednopunktowe poleca na przecięciu chromosomów rodzicielskich w wylosowanym punkcie k , w taki sposób, że pierwszy potomek otrzymuje k genów od pierwszego rodzica, natomiast resztę od drugiego, a w przypadku drugiego potomka odwrotnie. Przykład krzyżowania jednopunktowego przedstawiony został w tabeli 2.

Tabela 2
Krzyżowanie jednopunktowe.

Rodzic 1		Rodzic 2	
X ₁	X ₂	Y ₁	Y ₂
X ₃	X ₄	Y ₃	Y ₄
Potomek 1		Potomek 2	
X ₁	X ₂	Y ₁	Y ₂
Y ₃	Y ₄	X ₃	X ₄

źródło: opracowanie własne na podstawie (Anholcer, 2023)

W przypadku krzyżowania dwupunktowego, jeden z potomków otrzymuje geny do pierwszego punktu cięcia oraz po drugim punkcie cięcia od pierwszego rodzica, natomiast geny pomiędzy punktami cięcia od drugiego z rodziców. Analogicznie tworzony jest kolejny potomek.

Z kolei w przypadku krzyżowania z częściowym odwzorowaniem Patrially Mapped Crossover należy na początku wybrać dwa punkty cięcia np. 123|4567|89 oraz 862|1753|94. Następnie należy dokonać wymiany odcinkami pomiędzy punktami cięcia, w przypadku przytoczonego powyżej przykładu cyferki 4567 oraz 1753 zamieniają się stronami. Następnie należy utworzyć tabelę odwzorowań $1 \leftrightarrow 4, 7 \leftrightarrow 5, 5 \leftrightarrow 6, 3 \leftrightarrow 7$. Kolejny krok to ustalenie genów rodziców, które nie będą powodować konfliktów, w analizowanym przykładzie *2*|1753|89 oraz 8*2|4567|9*, a następnie w miejscach konfliktów ustawić geny zgodnie z tabelą odwzorowań, dzięki czemu uzyskany zostaje efekt 426|1753|89 oraz 832|4567|91.

Kolejny etap funkcjonowania algorytmów genetycznych to mutacja. Mutacja wprowadza losowe zmiany w chromosomach potomstwa, w celu utrzymania różnorodności genetycznej populacji. Najczęściej spotykaną metodą mutacji jest mutacja bitowa, w której każdy bit jest odwracany z prawdopodobieństwem p_m , które zwykle jest bardzo małe.

Algorytm kończy działanie po spełnieniu określonego warunku, np. osiągnięciu maksymalnej liczby iteracji T bądź znalezieniu rozwiązania spełniającego kryteria przystosowania $f(x^*)$.

(Anholcer, 2023) W odniesieniu do problemu komiwojażera, algorytm genetyczny ma obowiązek uwzględniać specyficzne wymagania z nim związane. W tym przypadku stosowane jest kodowanie kolejnościowe, w którym kod składa się z tylu znaków, ile miast występuje w zadaniu, a każdy ciąg kodowy jest przedstawicielem permutacji miasta, które należy odwiedzić. Przykładowo dane są cztery miasta, a rozwiązanie, w którym komiwojażer będzie miał możliwość odwiedzenia miast w kolejności 1,3,2,4 można zakodować jako

(1,3,2,4), dzięki czemu każde miasto zostanie odwiedzone wyłącznie jeden raz. Problem pojawia się w momencie próby krzyżowania oraz mutacji. W przypadku krzyżowania jednopunktowego na dwóch chromosomach (1,3,2,4) oraz (2,4,1,3) istnieje możliwość otrzymania potomstwa nie będącego poprawną permutacją miast np. (1,3,1,3). Jednym z rozwiązań tego problemu jest krzyżowanie porządkowe OX, które polega na przepisywaniu w każdym z ciągów początkowych elementów, które umieszczone są w odpowiednich pozycjach w ciągu drugim. Przykładowo w przypadku rodziców (1,3,2,4) oraz (2,4,1,3), wybrano miejsce krzyżowania pomiędzy punktem 2 oraz 3. Potomstwo wyglądać będzie w sposób następujący:

- Potomek 1: (1,3,1,3);
- Potomek 2: (2,4,2,4).

Kolejnym krokiem będzie wstawienie w odpowiednie miejsca elementów drugiego rodzica, tak aby zachować permutację. Po zmianie potomstwo wygląda następująco:

- Potomek 1: (1,3,4,2);
- Potomek 2: (2,4,1,3).

W przypadku mutacji odnosząc się do problemu komiwojażera należy również zastosować specjalistyczne podejście. Wyklucza się mutacje jednopunktową, stosując mutację polegającą na zamianie miejscami dwóch losowo wybranych elementów ciągu. Przykładowo dla ciągu (1,3,4,2) po zastosowaniu zamiany miejscami na pozycjach 2 oraz 4 otrzymany zostaje nowy ciąg (1,2,4,3).

W kontekście problemu TSP funkcja przystosowania stanowi odwrotność funkcji celu, czyli zamiast minimalizować długość trasy, dokonuje się maksymalizacji przystosowania, które jest odwrotnością trasy, co można przedstawić wzorem:

$$f(x) = \frac{1}{\text{długość rasy}} \quad (29)$$

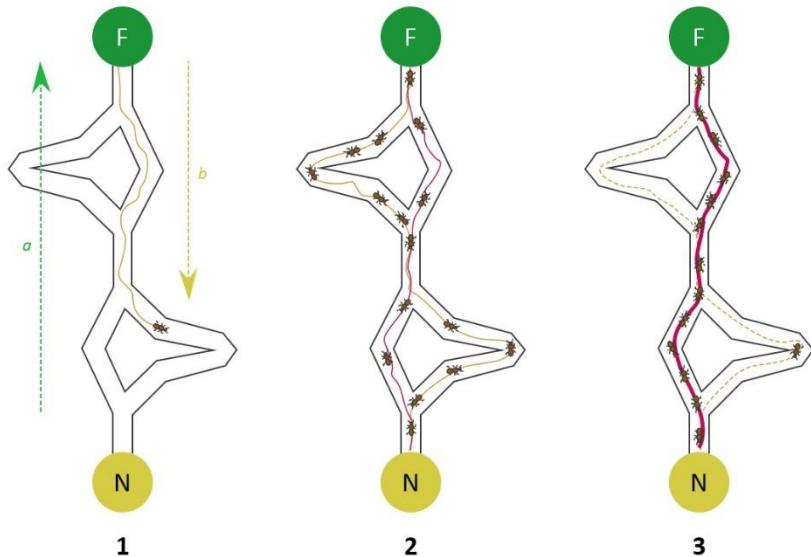
4.5. Algorytm mrówkowy

(Dorigo i Stützle, 2004) Algorytm mrówkowy (Ant Colony Optimization – ACO) to metaheurystyka, która nawiązuje do zachowania mrówek w kontekście poszukiwania najkrótszych ścieżek do źródła pokarmu za pomocą feromonów. (Błaszkiewicz, 2011) Algorytm zaproponowany został w 1991 roku w pracy doktorskiej Marco Dorigo, która była

zainspirowana opublikowaną w 1989 roku pracą o zachowaniu mrówek argentyńskich. (Dorigo i Stützle, 2004) Metaheurystyka rozwijana przez Marco Dorigo oraz Thomasa Stützle znalazła zastosowanie do rozwiązywania trudnych problemów optymalizacyjnych. Dokonując analizy algorytmu należy poznać podstawowe pojęcia, z pomocą których prościej będzie zrozumieć ideę algorytmu. Należą do nich:

- **Feromony** – substancje chemiczne wydzielane przez organizm, wywołujące specyficzne reakcje u osobników tego samego gatunku. W przypadku mrówek wykorzystywane są do komunikacji np. do oznaczenia ścieżki do źródła pokarmu. Pozostałe mrówki kierują się ścieżkami z wyczuwalnym silniejszym zapachem dzięki czemu wzmacniają go jeszcze bardziej. W kontekście algorytmu ACO dochodzi do komunikacji sztucznych mrówek, za pomocą sztucznego zapachu, symbolizującego numeryczną informację rozproszoną. Feromony wykorzystywane są do probabilistycznej konstrukcji rozwiązania problemu, natomiast ich wartości podlegają aktualizacji podczas działania ACO odzwierciedlając tym samym doświadczenia mrówek podczas poszukiwań;
- **Reguła prawdopodobieństwa** – mrówki dokonują wyboru drogi stosując prawdopodobieństwo zależne od ilości feromonów oraz innych heurystyk;
- **Aktualizacja feromonów** – feromony aktualizowane są na podstawie jakości odnalezionych rozwiązań, dzięki czemu zwiększana jest atrakcyjność dobrych ścieżek.

Algorytm swoją inspirację zaczerpnął z przeprowadzonego przez Deneubourga oraz jego współpracowników eksperymentu z podwójnym mostem, w którym te mrówki posiadały wybór dwóch ścieżek o różnych długościach, które prowadziły do źródła pokarmu. Na początku osobniki wybierały drogę losowo, jednak po pewnym czasie znaczna część mrówek zaczęła wybierać krótszą drogę. Krótsza ścieżka starała się znaczco oznaczona feromonami, co doprowadziło do zwiększenia jej popularności wyboru. Cały proces stanowi przykład autokatalizy, w której lokalne interakcje doprowadzają do globalnego wzorca.



Rysunek 28. Graficzne przedstawienie algorytmu mrówkowego.
[źródło: opracowanie własne na podstawie (Wikipedia, 2023)]

(Wikipedia, 2023) Doświadczenie w formie graficznej przedstawione zostało na rysunku 28. Symbol F znajdujący się w zielonym kółku symbolizuje gniazdo mrówek, natomiast N znajdujące się w żółtym kółku symbolizuje źródło pożywienia. W kroku pierwszym jedna z mrówek podąża ścieżką „a” do źródła pożywienia, aby następnie powrócić drogą „b” do gniazda zostawiając po sobie ślad w formie feromonu. Następnie pozostałe osobniki dokonują wyboru czterech ścieżek, jednak wzmacnienie intensywności feromonu czyni ścieżkę główną najatrakcyjniejszą i określa się ją jako najkrótszą drogę. Taki wybór sprawia, że w dłuższych odcinkach gęstość feromonów zanika i pozostałe mrówki wybierają najkrótszą drogę.

(Dorigo i Stützle, 2004) Modele matematyczne algorytmu mrówkowego opisują w jaki sposób mrówki dokonują wyboru odpowiedniej ścieżki oraz w jaki sposób aktualizowane są poziomy feromonu. Pierwszy etap stanowi prawdopodobieństwo wyboru ścieżki, w którym to mrówki wybierają trasę za pomocą ilości feromonów na krawędziach oraz heurystyk danej krawędzi, zazwyczaj odwrotności długości tzn. im krótsza krawędź tym wyższa heurystyka.

Prawdopodobieństwo $p_{ij}(t)$ wyboru krawędzi i oraz j przez mrówkę w węźle i w czasie t wyraża się za pomocą wzoru:

$$p_{ij}(t) = \frac{[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{k \in \text{allowed}} [\tau_{ik}(t)]^\alpha \cdot [\eta_{ik}]^\beta} \quad (30)$$

Gdzie:

- $\tau_{ij}(t)$ – symbolizuje ilość feromonów na krawędzi ij w czasie t ;
- η_{ij} – heurystyka dla krawędzi ij , najczęściej przyjmowana jako odwrotność procesu długości d_{ij} tej krawędzi: $\eta_{ij} = \frac{1}{d_{ij}}$;
- α – współczynnik wpływu feromonu na prawdopodobieństwo wyboru;
- β – współczynnik wpływu heurystyki na prawdopodobieństwo wyboru;
- „allowed” – zbiór krawędzi możliwych do wybrania przez mrówkę.

W kolejnym etapie następuje aktualizacja feromonów znajdujących się na krawędziach rozpoczynając od zmniejszenia ich poziomu z powodu parowania oraz dodania ich nowej ilości na podstawie tras przebytych przez mrówki. Proces parowania wyrażany jest wzorem:

$$\tau_{ij}(t+1) = (1 - \rho) \cdot \tau_{ij}(t) \quad (31)$$

Gdzie:

- $\tau_{ij}(t+1)$ – symbolizuje ilość feromonów na krawędzi ij po aktualizacji w czasie $t+1$;
- $\tau_{ij}(t)$ – symbolizuje ilość feromonów na krawędzi ij w czasie równym t ;
- ρ – współczynnik parowania feromonów, który zazwyczaj wynosi $\rho \in (0,1)$.

Następnie po procesie parowania dochodzi do aktualizacji feromonów poprzez dodanie nowej ich ilości pozostawionych przez przechodzące daną krawędzią mrówki, co opisywane jest wzorem:

$$\tau_{ij}(t+1) = (1 - \rho) \cdot \tau_{ij}(t) + \sum_{k=1}^m \Delta\tau_{ij}^k(t) \quad (32)$$

W równaniu (32) m symbolizuje liczbę mrówek, natomiast wyrażenie $\Delta\tau_{ij}^k(t)$ przedstawia ilość feromonów pozostawionych przez k -tą mrówkę na krawędzi ij . Ilość feromonów, jaką każda z mrówek pozostawiła na krawędzi, zależy od długości trasy, którą przebyła.

Algorytm mrówkowy został z powodzeniem zastosowany w kontekście rozwiązania problemu komiwojażera. ACO dla TSP wykorzystuje graf, w którym węzły reprezentują miasta, natomiast krawędzie symbolizują połączenia pomiędzy nimi. Każda krawędź posiada przypisaną wagę, która odpowiada odległości między miastami. Istnieje kilka głównych algorytmów mrówkowych dla problemu komiwojażera, do których zalicza się:

- **Ant System (AS)** – Jest to pierwszy algorytm mrówkowy, którego zadaniem jest inicjalizacja badań nad optymalizacją z pomocą kolonii mrówek. Stosuje równomierne rozmieszczenie feromonów na trasach, które wybrane zostały przez wszystkie mrówki, co z kolei doprowadziło do dobrej wydajności małych inicjalizacji problemu komiwojażera, jednak gorzej skalowało się do większych problemów;
- **Elitist Ant System (EAS)** – Wersja AS, której zadaniem jest wzmacnienie śladu feromonu na najlepszej odnalezionej trasie, co powodowało przyśpieszenie konwergencji algorytmu;
- **Rank-based Ant System (ASrank)** – jego zadaniem jest różnicowanie ilości feromonów na podstawie ich rankingu w bieżącej iteracji;
- **Max-Min Ant System (MMAS)** – wprowadza ograniczenia na minimalne oraz maksymalne wartości feromonów, dzięki czemu unika się przedwczesnej konwergencji i stagnacji;
- **Ant Colony System (ACS)** – algorytm wprowadza lokalne aktualizacje feromonów podczas opracowywania trasy oraz globalne aktualizacje dokonywane na najlepszej trasie, co poprawia wydajność na różnych instancjach problemu komiwojażera.

Algorytmy mrówkowe charakteryzują się ogromną skutecznością w rozwiązywaniu problemu komiwojażera, szczególnie w przypadku, gdy są połączone z technikami optymalizacji lokalnej.

4.6. Algorytm Christofidesa

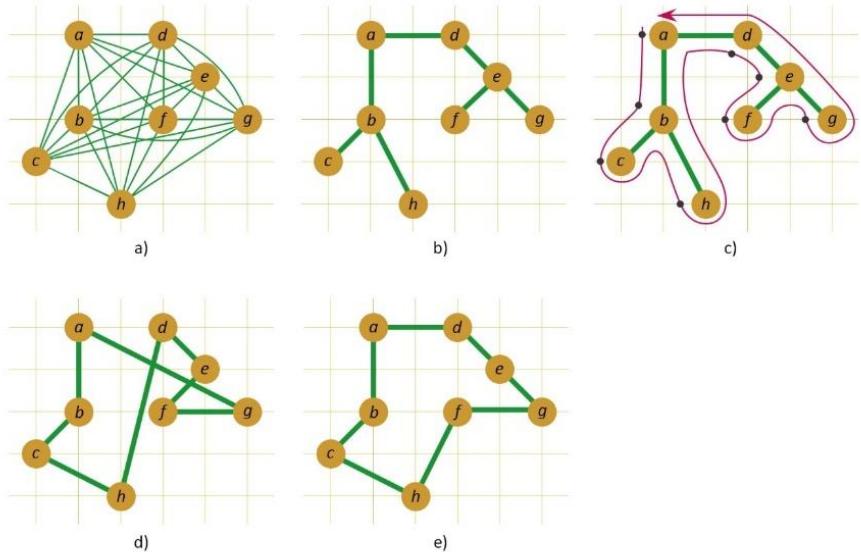
(Cormen, Leiserson, Rivest i Stein, Introduction to Algorithms, 2009) Algorytm Christofidesa jest przedstawicielem algorytmów aproksymacyjnych dla problemu komiwojażera, gdzie odległości spełniają nierówności trójkąta. Celem algorytmu jest odnalezienie cyklu Hamiltona, którego koszt nie będzie gorszy niż 1,5 razy koszt optymalnego cyklu oraz musi spełniać warunek nierówności trójkąta, a także wagi krawędzi nie mogą być ujemne.

W odróżnieniu od heurystyk czy algorytmów genetycznych od początku gwarantuje on jakość rozwiązania, która bywa wystarczająca. Proces działania algorytmu składa się z kilku kroków. Na początku należy odnaleźć minimalne drzewo rozpinające dla grafu pełnego $G = (V, E)$, który zawiera funkcję kosztu $c(u, v)$, a samo drzewo można odnaleźć z pomocą algorytmu Prima, bądź Kruskala. Samo drzewo jest podgrafem $T \subseteq G$ zawierającym wszystkie wierzchołki V oraz minimalną sumę kosztów krawędzi. W kolejnym kroku następuje identyfikacja wierzchołków o nieparzystym stopniu w zbiorze T , jednak z uwagi na właściwość drzew liczba wierzchołków jest parzysta. Po dokonaniu identyfikacji nieparzystych wierzchołków algorytm wyszukuje minimalne doskonałe dopasowanie¹¹ M , tworząc tym samym multigraf¹² G' będący grafem Eurela. Następnie zadaniem algorytmu jest przejście przez cały cykl Eurela w grafie G' oraz dokonanie przekształcenia go w cykl Hamiltona za pomocą pominięcia powtarzających się wierzchołków. Nierówność trójkąta zapewnia, że taki proces nie zwiększa całkowitego kosztu trasy.

Algorytm Christofidesa oparty jest na nierówności trójkąta, stanowiącej dla dowolnych wierzchołków $u, v, w \in V$ koszt bezpośredniego przejścia z wierzchołka u do wierzchołka w , który jest mniejszy lub równy sumie kosztów przejścia z wierzchołka u do v oraz z v do w , co można wyrazić za pomocą wzoru $c(u, w) \leq c(u, v) + c(v, w)$. Właściwość ta zapewnia algorytmowi możliwość stworzenia cyklu Hamiltona o koszcie nie gorszym niż 1.5 razy kosztu cyklu optymalnego. Na rysunku 29 zaprezentowane zostało działanie algorytmu na przykładzie grafu.

¹¹Minimalne doskonałe dopasowanie (ang. perfect matching) to zbiór krawędzi w grafie, który łączy wszystkie wierzchołki w pary, tak aby każdy wierzchołek był dokładnie raz końcem jednej z krawędzi, a suma wag tych krawędzi była najmniejsza możliwa.

¹² Multigraf - graf, w którym między dowolnymi dwoma wierzchołkami może istnieć więcej niż jedna krawędź, a także mogą występować pętle, czyli krawędzie łączące wierzchołek sam ze sobą.



Rysunek 29. Działanie algorytmu Christofidesa

[źródło: opracowanie własne na podstawie (Cormen, Leiserson, Rivest i Stein, *Wprowadzenie do algorytmów*, 2018)]

Powyżej umieszczony rysunek 29 przedstawia krok po kroku etapy algorytmu Christofidesa na przykładzie grafu, przechodząc przez kolejne etapy tworzenia minimalnego drzewa rozpinającego, doskonałego dopasowania oraz przejście do cyklu Hamiltona. Podpunkt a przedstawia pełny graf nieskierowany G , w którym wierzchołki są punktami na przecięciu linii siatki całkowitoliczbowej. Kolejny podpunkt przedstawia minimalne drzewo rozpinające T dla grafu pełnego G , wyznaczone za pomocą algorytmu Prima. Wierzchołek a stanowi korzeń. Podpunkt c przedstawia przejście drzewa T metodą preorder, tworząc cykl Hamiltona. Wierzchołki odwiedzane są w kolejności a,b,c,h,d,e,f,g. Kolejna część rysunku przedstawia minimalne doskonałe dopasowanie M dla wierzchołków o nieparzystym stopniu w T . Ostatni element stanowi połączenie krawędzi drzewa T dopasowania M , tworząc multigraf Eurela, z którego cykl Eurelowski jest przekształcany w cykl Hamiltona poprzez omijanie powtarzających się wierzchołków.

4.7. Algorytm Little'a

(Anholcer, 2023) Algorytm Little'a to efektywne podejście do rozwiązywania problemu komiwojażera, którego celem jest minimalizacja kosztu całkowitego podróży pomiędzy miastami, powracając na końcu do punktu wyjścia. (EconPapers, 2023) Został opracowany w 1963 roku przez Johna D. C. Little'a, Katta G. Murty'ego, Dura W. Sweeney'a oraz Caroline Karel. (Anholcer, 2023) Polega on na systematycznym redukowaniu macierzy kosztów

podróży pomiędzy miastami, gdzie każda z komórek macierzy reprezentuje koszt podróży pomiędzy miastem i oraz j. Redukcja osiągana jest poprzez odjęcie najmniejszego elementu z każdego wiersza oraz kolumny, co niesie za sobą obniżkę kosztów bez zmiany relatywnej efektywności różnych tras.

Proces algorytmu składa się z trzech kroków: redukcji macierzy kosztów, wyboru i zakazania ścieżek oraz rekurencyjnego podziału problemu. Na wstępie algorytm redukuje macierz celem zmniejszenia kosztu ogólnego bez zmiany optymalnego rozwiązania poprzez odjęcie najmniejszego elementu w każdym wierszu. Proces ten opisany jest wzorem:

$$a_i = \min_j(c_{ij}), \quad i = 1, \dots, n \quad (33)$$

Gdzie:

- c_{ij} – symbolizuje koszt podróży pomiędzy miastami i oraz j;
- a_i – minimalna wartość w wierszu i.

Kolejny krok to redukcja macierzy poprzez odjęcie najmniejszej wartości z każdej kolumny, co zostało opisane wzorem:

$$b_j = \min_i(c_{ij} - a_i), \quad j = 1, \dots, n \quad (34)$$

We wzorze (34) b_j reprezentuje minimalną wartość w kolumnie j po przeprowadzeniu redukcji wierszy. W kolejnym etapie wybierane są ścieżki, które nie tworzą cykli, minimalizując przy tym koszty podróży. W celu uniknięcia cykli mogących zakłócić rozwiązanie, algorytm Little'a przypisuje niektórym ścieżkom nieskończenie duże koszty. Po dokonaniu wyboru ścieżki pomiędzy miastem i oraz j, koszt wykluczenia obliczany jest jako suma najmniejszych wartości z pozostałych elementów wiersza oraz kolumny, z której ścieżka została wybrana, co można zapisać wzorem:

$$d_{ij} = \min_{\substack{k \in \{1, \dots, n\} \\ k \neq i}} \{c_{ik}\} + \min_{\substack{l \in \{1, \dots, n\} \\ l \neq j}} \{c_{lj}\}, \quad c_{ij} = 0 \quad (35)$$

We wzorze (35) d_{ij} symbolizuje oszacowany koszt zamknięcia ścieżki pomiędzy miastami i oraz j, natomiast k to indeksy pozostałych miast. Kolejnym etapem jest podział problemu na mniejsze zarządzalne segmenty, eliminując tym samym nieoptimalne ścieżki, skupiając uwagę wyłącznie na najbardziej obiecujących trasach. Proces dzieli się na coraz mniejsze fazy, w których każdy kolejny podział jest traktowany jako nowy problem komiwojażera do rozwiązania, do momentu osiągnięcia minimalnego kosztu podróży.

Dla zobrazowania działania algorytmu Little'a przedstawiony zostanie przykład zadania. W tabeli 1 znajduje się macierz kosztów. Dodatkowy wiersz oraz kolumna zawierają koszty redukcji obliczone za pomocą równań numer X oraz X. Przykładowo dla przykładu a_1 minimalny koszt wynosi $a_1 = \min\infty, 1, 7, 2 = 1$. W przypadku kolumn obliczane jest minimum różnic kosztów dla tras oraz kosztów redukcji w wierszach np. dla b_3 koszt wynosi: $b_3 = \min 7 - 1, 7 - 4, \infty - 3, 4 - 2 = 2$

Tabela 3.

Początkowe koszty dla tras i koszty redukcji dla podzbioru D_0 .

D_o	M	W1	W2	W3	a_i
M	∞	1	7	2	1
W1	6	∞	7	4	4
W2	8	3	∞	5	3
W3	2	7	4	∞	2
b_j	0	0	2	0	

źródło: opracowanie własne na podstawie (Anholcer, 2023)

W kolejnym kroku należy dokonać redukcji kosztów odejmując od każdego kosztu w tabeli koszty redukcji wyznaczone dla wiersza oraz kolumny zgodnie ze wzorem (36):

$$c'_{ij} = c_{ij} - a_i - b_j, \quad i = 1, \dots, n, \quad j = 1, \dots, n \quad (36)$$

Przykładowo dla tras z pierwszego wierszu wyniki wynoszą: $c'_{11} = \infty - 1 - 0 = \infty$, $c'_{12} = 1 - 1 - 0 = 0$, $c'_{13} = 7 - 1 - 2 = 4$, $c'_{14} = 2 - 1 - 0 = 1$. Wyniki redukcji kosztów przedstawione zostały w tabeli 4.

Tabela 4.

Zredukowana macierz kosztów.

D_o	M	W1	W2	W3
M	∞	0	4	1
W1	2	∞	1	0
W2	5	0	∞	2
W3	0	5	0	∞

źródło: opracowanie własne na podstawie (Anholcer, 2023)

Kolejnym krokiem jest oszacowanie kosztów redukcji, które dotychczas wynosiło 0. Oszacowanie obliczane jest według wzoru:

$$\Delta w = \sum_{i=1}^n a_i + \sum_{j=1}^n b_j \quad (37)$$

Dla przykładu z tabeli 2 oszacowanie wynosi: $w = 1 + 4 + 3 + 2 + 0 + 0 + 2 + 0 = 12$. Z uwagi na fakt, że zbiór D_0 nie jest pusty, a macierz kosztów ma wymiary 4×4 , oznacza to, że zbiór ten nie jest jednoelementowy i nie można zamknąć go na tym etapie, ponieważ jest jedynym aktywnym zbiorem i ulega podziałowi. Aby dokonać podziału należy wyznaczyć koszt rezygnacji z poszczególnych tras za pomocą wzoru 37. Przykładowo dla trasy $\langle M, W1 \rangle$ koszt wynosi: $\min\infty, 4, 1 + \min\infty, 0, 5 = 1 + 0 = 1$.

Tabela 5.

Koszty rezygnacji z podzbioru D_0 .

D_o	M	W1	W2	W3
M	∞	1	4	1
W1	2	∞	1	2
W2	5	2	∞	2
W3	2	5	1	∞

źródło: opracowanie własne na podstawie (Anholcer, 2023)

Koszty podziału przedstawione zostały w tabeli 5, ich najwyższy wynik wynosi 2, co odpowiada trasom $\langle W1, W3 \rangle$, $\langle W2, W1 \rangle$ oraz $\langle W3, M \rangle$. Z kolei podzbiór D_1 powstał jako zbiór cykli zawierający wyłącznie trasę $\langle W1, W3 \rangle$. Aby utworzyć macierz dla zbioru D_1 należy wykreślić wiersz W1, ponieważ z tego miejsca nie można już udać się nigdzie indziej, a także kolumnę W3, ze względu na fakt, że nie można już do niej przyjechać z żadnej miejscowości. Powstała nowa macierz przedstawiona została w tabeli 6.

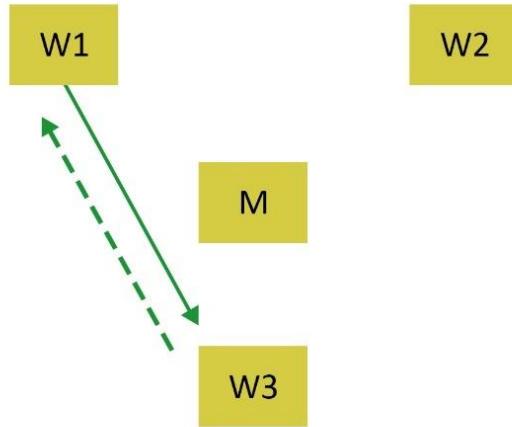
Tabela 6.

Macierz kosztów po wykreśleniu trasy $\langle W1, W3 \rangle$.

D_1	M	W1	W2
M	∞	0	4
W2	5	0	∞
W3	0	5	0

źródło: opracowanie własne na podstawie (Anholcer, 2023)

Kolejnym krokiem, który należy wykonać jest zablokowanie trasy $\langle W3, W1 \rangle$, w celu niedopuszczenia do powstania cyklu pomiędzy dwoma miejscowościami. Proces ten przedstawiony został na rysunku 30, gdzie przerywana linia symbolizuje zablokowaną linię.



Rysunek 30. Blokada trasy $\langle W3, W1 \rangle$ w celu niedopuszczenia do powstania cyklu
[źródło: opracowanie własne na podstawie (Anholcer, 2023)]

Aby dokonać blokady trasy $\langle W3, W1 \rangle$ należy wstawić ∞ w miejsce ich kosztu w tabeli, co zostało przedstawione w tabeli 7.

Tabela 7.

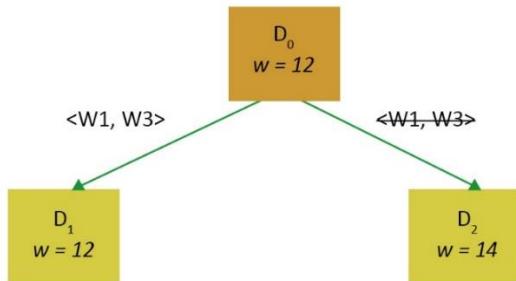
Macierz kosztów dla podzbioru D_1 po zablokowaniu trasy $\langle W3, W1 \rangle$

D_1	M	W1	W2	a_i
M	∞	0	4	0
W2	5	0	∞	0
W3	0	5	0	0
b_j	0	0	0	

źródło: opracowanie własne na podstawie (Anholcer, 2023)

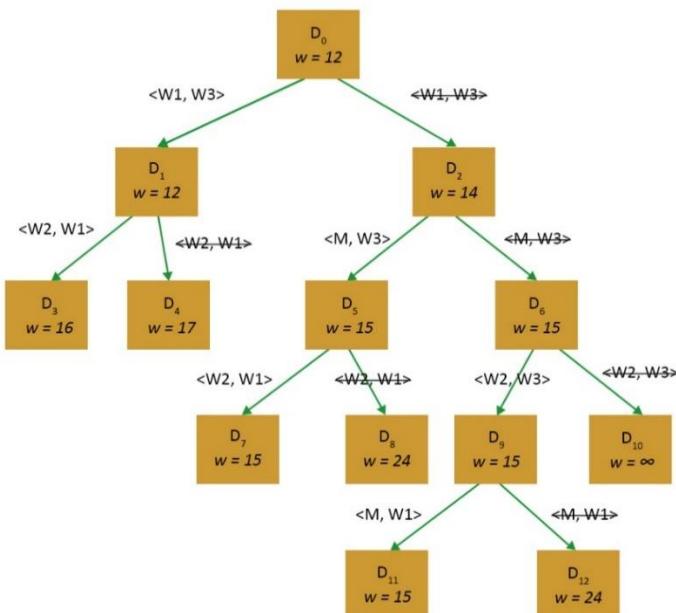
W kolumnie a_i oraz wierszu b_j zamieszczone zostały koszty po redukcji. Ze względu na fakt, że ich suma wynosi $\Delta w(D_1) = 0$, natomiast podzbiór D_1 powstał z podzbioru D_0 oszacowany koszt dla niego jest równy $w(D_1) = \Delta w(D_0) + \Delta w(D_1) = 12 + 0 = 12$. Aby wyznaczyć oszacowanie dla podzbioru D_2 należy dodać do wcześniej obliczonego oszacowania koszt rezygnacji z trasy $\langle W1, W3 \rangle$, który wynosił 2. Zatem otrzymane zostają wartości $w(D_2) = w(D_0) + d_{W1,W3} = 12 + 2 = 14$. Wyniki, które zostały dotychczas

uzyskane zaprezentowane zostały na drzewie rozwiązań, które zostało przedstawione na rysunku 31. Kolor pomarańczowy symbolizuje zbiory zamknięte, natomiast żółty otwarte.



Rysunek 31. Drzewo rozwiązań
[źródło: opracowanie własne na podstawie (Anholcer, 2023)]

Proces podziałów oraz redukcji kontynuowany jest, aż do momentu, gdy uzyskane zostaną drzewa jednocołowych zbiorów. Ze względu na złożoność tego procesu poniżej przedstawione zostało ostateczne drzewo rozwiązań, które zilustrowane zostało na rysunku 32. W wyniku analizy powstały trzy zbiory jednoelementowe, do których zalicza się D₃ (cykl (M, W2, W1, W3, M), koszt równy 16), D₇ (cykl (M, W3, W2, W1, M), koszt równy 15) oraz D₁₁ (cykl (M, W1, W2, W3, M), koszt równy 15). Ze względu na najniższy koszt zbiory D₇ oraz D₁₁ są optymalne.



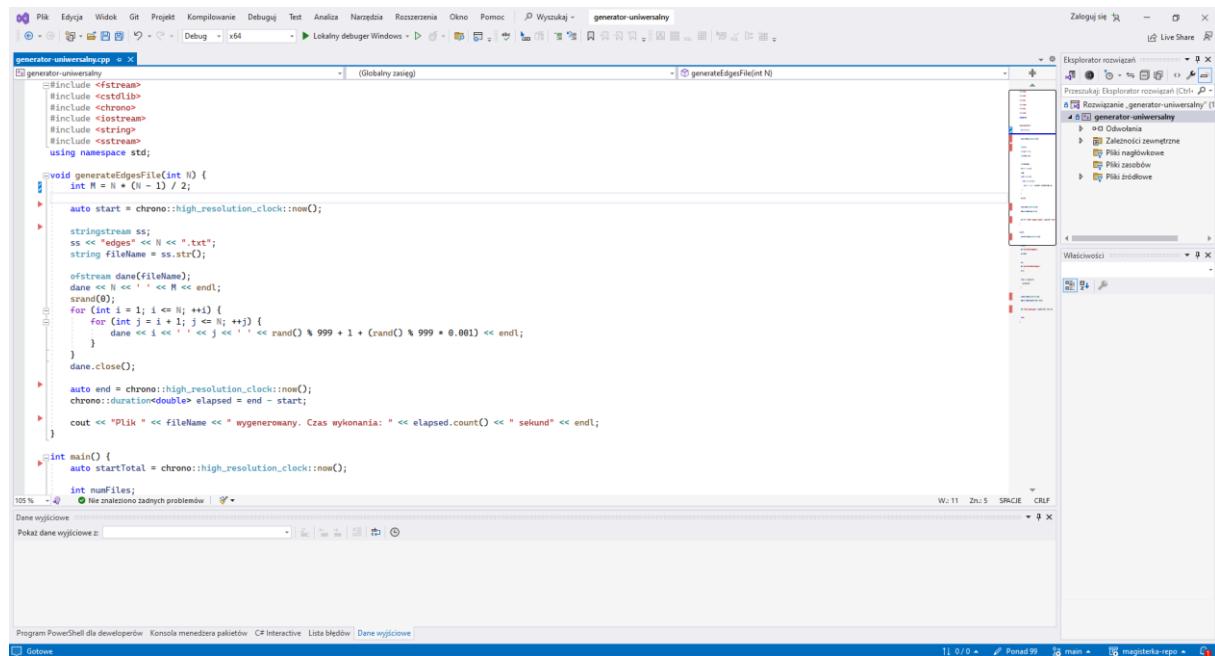
Rysunek 32. Ostateczne zamknięte drzewo rozwiązań
[źródło: opracowanie własne na podstawie (Anholcer, 2023)]

5. Przedstawienie środowiska pracy

Rozdział piąty stanowi zapoznanie z środowiskiem pracy, w którym wykonana została praca dyplomowa. Zawiera przedstawienie środowiska Visual Studio, GitHub, Jupyter Notebook oraz Matlab. Omówione zostały również języki, w których zaimplementowane zostały programy potrzebne do przeprowadzenia analizy wybranych algorytmów rozwiązania problemu komiwojażera.

5.1. Visual Studio Community 2022

(Learn Microsoft, 2024) Visual Studio Community 2022 stanowi darmowe środowisko programistyczne oferujące różnorakie narzędzia do implementacji wszelkiego rodzaju aplikacji na platformy Android, Windows, iOS, a także aplikacje internetowe. Cechą wyróżniającą to środowisko jest obsługa wielu języków programowania m.in. C++, C#, Python, TypeScript oraz inne.



Rysunek 33. Interfejs programu Visual Studio Community 2022
[źródło: opracowanie własne]

5.2.Język C++

(Zieliński, Podstawy programowania w języku C++, 2013) Język C++ to język programowania poziomu wysokiego, umożliwiający tworzenie kodu źródłowego, który następnie kompilowany jest do kodu maszynowego. Proces kompilacji obejmuje m.in. analizę składniową oraz semantyczną, która z kolei prowadzi do utworzenia kodu obiektowego. Następnie kod jest łączony z dodatkowymi bibliotekami, w celu stworzenia pliku wykonalnego. Język C++ łączy cechy programowania proceduralnego oraz obiektowego. Wpiera różne typy danych m.in. liczby całkowite, zmienoprzecinkowe oraz konstrukcje kontrolne takie jak pętle czy instrukcje warunkowe. (Wikipedia, 2024) C++ jest standaryzowany przez Międzynarodową Organizację Normalizacyjną (ISO). Najnowsza stabilna wersja standardu to C++ 20, opublikowana w grudniu 2020 roku. Rysunek X przedstawia przykładowy kod programu wykonany w języku C++.

5.3. Jupyter Notebook

(VanderPlas, 2023) Jupyter Notebook stanowi interaktywne środowisko webowe umożliwiające tworzenie oraz udostępnianie dokumentów zawierających m.in. żywy kod, równania wraz z wizualizacjami, czy też zawierać narracyjny tekst. Środowisko wykorzystuje architekturę klient-serwer, dzięki czemu obsługuje wiele języków programowania m.in. Python, R, Scala oraz inne. Oprogramowanie to daje możliwość śledzenia wyników działania programu na żywo. Jupyter Notebook szczególną popularność zyskał w dziedzinach analizy danych, uczenia maszynowego, czy też w środowisku edukacyjnym do tworzenia obliczeń oraz dokumentacji edukacyjnej wymagającej połączenia kodu tekstu oraz grafiki. Rysunek 34 przedstawia interfejs omawianego środowiska pracy.

```

In [3]: 1 import time
2 from math import fabs
3 from typing import List, Tuple
4
5 class Edge:
6     def __init__(self, v1: int, v2: int, weight: float):
7         self.v1 = v1
8         self.v2 = v2
9         self.weight = weight
10
11     def __lt__(self, other):
12         return self.weight < other.weight
13
14 class Vertex:
15     def __init__(self):
16         self.row = 0
17         self.v1 = 0
18         self.v2 = 0
19
20 def read_data(filename: str) -> Tuple[int, int, List[Edge]]:
21     edges = []
22     with open(filename, 'r') as file:
23         N, M = map(int, file.readline().split())
24         for _ in range(N):
25             v1, v2, weight = map(int, file.readline().split())
26             edges.append(Edge(int(v1), int(v2), weight))
27     return N, M, edges
28
29 def ncycle(vs: List[Vertex], e: Edge) -> bool:
30     v1, v2 = e.v1, e.v2
31     v1 = vs[v1].v1
32     v2 = vs[v2].v1
33     while v1 != v2 and vs[v1].row == 2:
34         if vs[v1].v1:
35             v1 = vs[v1].v1
36         else:
37             v2 = vs[v1].v2
38     return v1 != v2
39
40 def build(N: int, edges: List[Edge]) -> Tuple[float, float]:
41     vs = [Vertex() for _ in range(N + 1)]
42     edges.sort()
43     w = 0.0
44     L = 0.0
45
46     i = 1
47     for t in edges:
48         if vs[t.v1].row == 2 or vs[t.v2].row == 2:
49             if vs[t.v1].row + vs[t.v2].row <= 1:
50                 if vs[t.v1].v1:
51                     vs[t.v1].v1 = t.v1
52                 else:
53                     vs[t.v1].v2 = t.v1
54                 if vs[t.v2].v1:
55                     vs[t.v2].v1 = t.v2
56                 else:
57                     vs[t.v2].v2 = t.v2
58             vs[t.v1].row += 1
59             vs[t.v2].row += 1
60             w += t.weight
61             L += fabs(t.v1 - t.v2)

```

Rysunek 34. Przedstawienie interfejsu Jupyter Notebook.
[źródło: opracowanie własne]

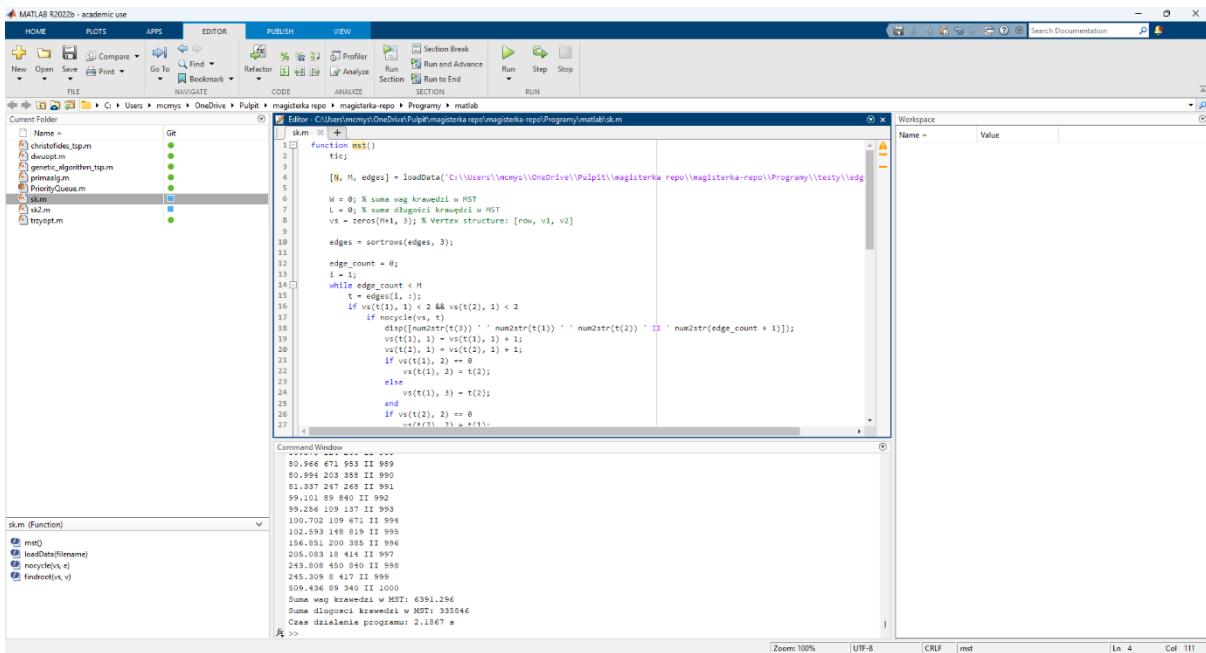
5.4. Język Python

(Martelli, Martelli Ravenscroft, Holden i McGuire, 2023) Python jest wysokopoziomowym, interpretowanym językiem programowania ogólnego przeznaczenia. Cechuje się wysoką produktywnością w analizie, projektowaniu, prototypowaniu, kodowaniu, testowaniu, debugowaniu, dostrajaniu, dokumentowaniu oraz utrzymaniu. Dzięki swojej prostocie, regularności, bogatej bibliotece standardowej oraz wielu dostępnych pakietom jest prosty w nauce. (Wikipedia, 2024) Język Python wspiera różne wzorce programowania m.in. obiektowe, imperatywne oraz w znacznie mniejszym stopniu funkcyjny. Aktualnie najnowsza stabilna wersja języka to 3.12.4 z czerwca 2024 roku.

5.5. MatLab

(Gilat, 2016) MatLab to potężny język obliczeń technicznych stosowany szczególnie do dokonywania obliczeń matematycznych, modelowania wraz z symulacją oraz analizy i przetwarzania danych, a także do rozwoju algorytmów. Oprogramowanie zawiera podstawowe funkcje służące do rozwiązywania typowych problemów. Zawiera również zestawy narzędzi, przeznaczonych do rozwiązania konkretnych typów problemów. Szczególną popularność zyskał w środowisku akademickim oraz przemyśle, ze względu na konieczność

przeprowadzania badań, rozwoju oraz projektowania. Rysunek 35 przedstawia interfejs programu MatLab.



Rysunek 35. Przedstawienie interfejsu MatLab
[źródło: opracowanie własne]

5.6. System kontroli wersji GitHub

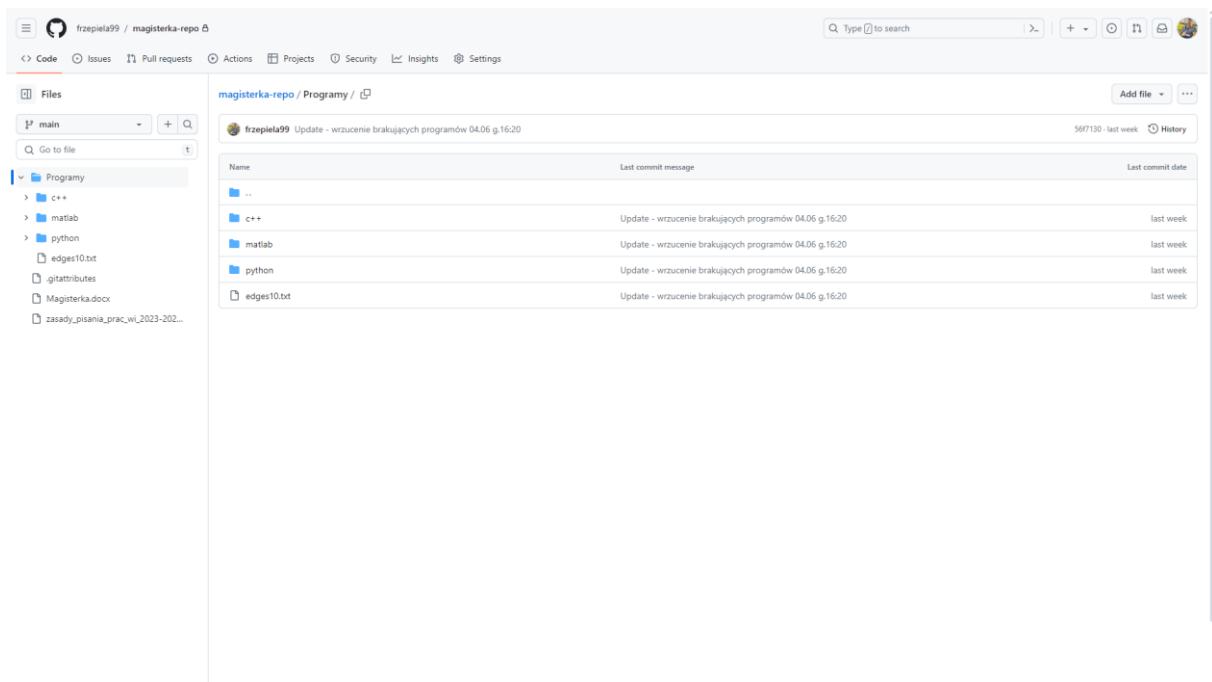
(Bell i Brent, 2015) GitHub stanowi najpopularniejszy system kontroli wersji¹³, służący do przechowywania kodu źródłowego, a także istnieje możliwość współpracy pomiędzy użytkownikami nad projektami programistycznymi. Platforma zapewnia bezpieczeństwo pracy z kodem źródłowym poprzez tworzenie kopii zapasowych kodu, co daje użytkownikom możliwość przywrócenia poprzedniej wersji kodu, w przypadku błędnych modyfikacji. Systemy kontroli wersji takie jak GitHub oferują szereg korzyści, które mają wpływ na zarządzanie projektami programistycznymi. Zalicza się do nich m.in.:

- Archiwizację historii – systemy kontroli wersji pozwalają na dostęp do wcześniejszych wersji projektu, dzięki czemu można sprawnie przeprowadzić analizę poprawności wcześniej podjętych decyzji, a także obserwację ewolucji poszczególnych składników projektu;

¹³ System kontroli wersji - oprogramowanie monitorujące modyfikacje, głównie w kodzie źródłowym. Wspiera programistów w integracji zmian wprowadzanych przez różnych uczestników projektu w różnych okresach.

- Przypisanie autorstwa – umożliwiają zidentyfikowanie twórców poszczególnych fragmentów kodu, a także komentarzy, co przekłada się na lepszą współpracę w zespole oraz rozliczenie z wykonanej pracy;
- Bezpieczeństwo eksperymentów – systemy kontroli wersji zapewniają możliwość przeprowadzania testów nowych rozwiązań w odizolowanym środowisku, chroniąc tym samym stabilne wersje oprogramowania. W przypadku, gdy próby odnoszą oczekiwany sukces, istnieje możliwość łatwej integracji z główną linią projektu.

Rysunek 36 przedstawia interfejs repozytorium założonego na platformie GitHub, wykonane na potrzeby realizacji pracy dyplomowej.



Rysunek 36. Przedstawienie interfejsu repozytorium GitHub.
[źródło: opracowanie własne]

GitHub oferuje możliwość przesyłania plików do repozytorium za pomocą wiersza poleceń, poprzez przeciągnięcie pliku za pomocą przeglądarki internetowej lub za pomocą aplikacji komputerowej GitHub Destkop. (GitHub Docs, 2024) Aplikacja GitHub Desktop działa z poziomu pulpitu komputera. Umożliwia wykonanie podstawowych operacji Git m.in. rejestrowanie zmian, tworzenie nowych gałęzi, przesyłanie najnowszych zmian do repozytorium oraz ich pobieranie z hostingu, bez konieczności używania wiersza poleceń.

6. Implementacja komputerowa

Rozdział szósty stanowi przedstawienie komputerowej implementacji wybranych algorytmów rozwiązujących problem komiwojażera w językach C++, Python oraz MatLab. W celu zweryfikowania poprawności działania algorytmów, testy przeprowadzone zostały na tych samych danych wygenerowanych do pliku przez zaimplementowany generator liczb losowych.

6.1. Generator grafów losowych

W celu przeprowadzenia badań i efektywności kluczowe jest przeprowadzenie analizy różnych algorytmów na dokładnie tych samych danych. W tym celu opracowany został uniwersalny generator grafów losowych. Do zadań generatora należy tworzenie wierzchołków oraz krawędzi, a także przypisywanie im losowych wag. Program opracowany został za pomocą języka C++.

Na początku program pyta użytkownika o liczbę plików oraz ilość wierzchołków do wygenerowania dla każdego z plików. Fragment kodu odpowiedzialny za tą funkcję został przedstawiony na rysunku 37.

```
int numFiles;
cout << "Podaj liczbę plików do wygenerowania: ";
cin >> numFiles;

int N;
cout << "Podaj liczbę wierzchołków dla każdego pliku: ";
cin >> N;

for (int i = 1; i <= numFiles; ++i) {
    generateEdgesFile(N);
}
```

Rysunek 37. Fragment kodu odpowiedzialny za interakcje z użytkownikiem
[źródło: opracowanie własne]

Proces generowania wierzchołków oraz krawędzi opiera się na dwóch zagnieżdżonych pętlach. Pierwsza z nich rozpoczyna się od wartości numer 1, natomiast kończy się na wprowadzonej przez użytkownika liczbie. Z kolei druga z nich swoje działanie rozpoczyna od wartości równej bieżącej wartości z pierwszej pętli, zwiększonej o jeden i również kończy się na liczbie wprowadzonej przez użytkownika. Następnie wartości

zmiennych iterowanych w tych pętlach są zapisywane do pliku jako krawędzie. Fragment kodu odpowiedzialny za generowanie wierzchołków oraz krawędzi przedstawiony został na rysunku 38.

```
for (int i = 1; i <= N; ++i) {
    for (int j = i + 1; j <= N; ++j) {
        dane << i << ' ' << j << ' ' << rand() % 999 + 1 + (rand() % 999 * 0.001) << endl;
    }
}
```

Rysunek 38. Fragment kodu odpowiedzialny za generowanie krawędzi i wierzchołków
[źródło: opracowanie własne]

Wagi krawędzi w przedstawianym generatorze generowane są losowo i mogą być to wartości całkowite bądź zmiennoprzecinkowe. W celu wygenerowania liczby skorzystano z biblioteki cstdlib, która umożliwia wykorzystanie funkcji rand(). Zakres losowania ustalony został w przedziale od 1 do 999. Fragment kodu odpowiedzialny za określenie zakresu generowania wag krawędzi został przedstawiony na rysunku 39.

```
rand() % 999 + 1 + (rand() % 999 * 0.001)
```

Rysunek 39. Fragment kodu odpowiedzialny za generowanie wag krawędzi
[źródło: opracowanie własne]

Zapis wygenerowanych danych do pliku umożliwia wykorzystanie biblioteki fstream. Jednak, aby operacja mogła przebiec pomyślnie, w pierwszej kolejności należy utworzyć strumień pliku. Poniżej przedstawiony rysunek 40 przedstawia fragment kodu odpowiedzialny za utworzenie strumienia pliku.

```
ofstream dane(fileName);
dane << N << ' ' << M << endl;
srand(0);
for (int i = 1; i <= N; ++i) {
    for (int j = i + 1; j <= N; ++j) {
        dane << i << ' ' << j << ' ' << rand() % 999 + 1 + (rand() % 999 * 0.001) << endl;
    }
}
dane.close();
```

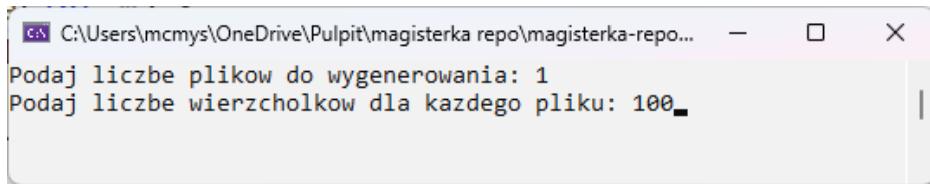
Rysunek 40. Fragment kodu odpowiedzialny za strumień pliku.
[źródło: opracowanie własne]

Nazwa pliku jest generowana poprzez przechwycenie podanej przez użytkownika ilości wierzchołków do wygenerowania. Przykładowo, jeżeli użytkownik wprowadzi 100 wierzchołków, to nazwa pliku będzie edges100.in. Fragment kodu odpowiedzialny za nazwę pliku przedstawiony został na rysunku 41.

```
stringstream ss;
ss << "edges" << N << ".in";
string fileName = ss.str();
```

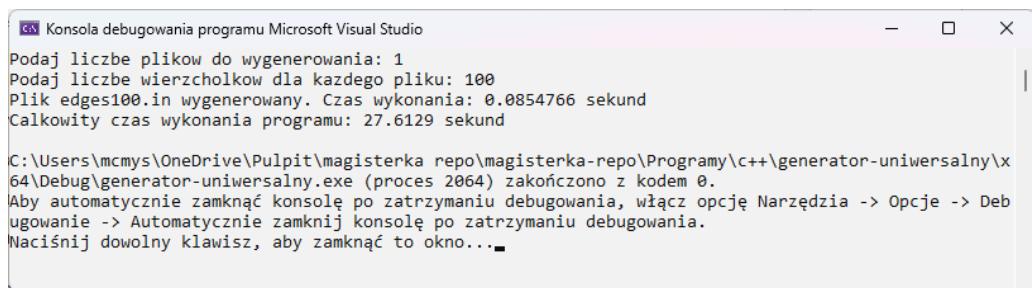
Rysunek 41. Fragment odpowiedzialny za nazewnictwo wygenerowanego pliku
[źródło: opracowanie własne]

Opracowany program generatora uruchamiany jest z poziomu terminala. Na wstępie użytkownik zostaje zapytany o ilość plików do wygenerowania. Po podaniu liczby program wyświetla kolejne pytanie o liczbę wierzchołków dla każdego pliku. Proces ten przedstawiony został na rysunku 42.



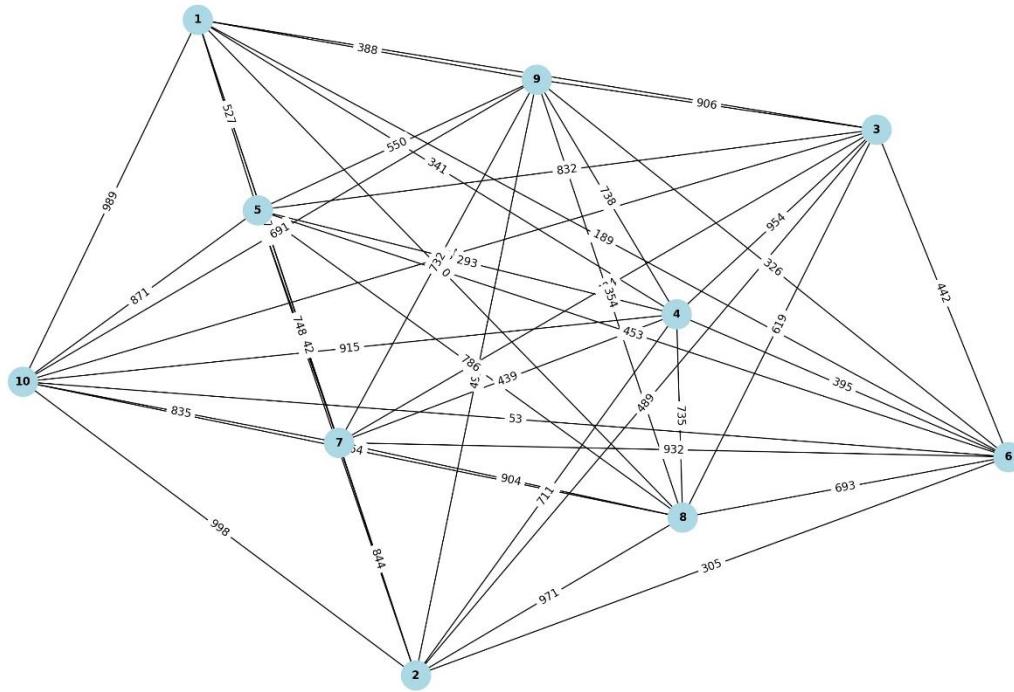
Rysunek 42. Proces działania programu po uruchomieniu
[źródło: opracowanie własne]

Po podaniu zarówno liczby plików do wygenerowania, jak i ilości wierzchołków program rozpoczyna tworzenie pliku zawierającego dane wejściowe. Użytkownik uzyskuje informację o wygenerowaniu pliku oraz czasie w jakim dane wejściowe zostały sporządzone z wykorzystaniem opracowanego generatora. Wyświetlany jest czas całkowity działania programu. Wszelkie informacje, które zostały zwrócone przedstawione zostały na rysunku 43.



Rysunek 43. Informacje zwrócone przez program
[źródło: opracowanie własne]

W przykładowo wygenerowanym pliku zawierającym dane wejściowe znajduje się informacja o ilości wierzchołków oraz krawędzi grafu. Z kolei pozostałe linie zawierają losowe dane takie jak numer pierwszego oraz drugiego wierzchołka oraz waga krawędzi. Przykładowy graf wygenerowany dla 10 wierzchołków przedstawiony został na rysunku 44.



Rysunek 44. Wizualizacja wygenerowanego dla 10 wierzchołków grafu
[źródło: opracowanie własne]

6.2. Implementacja algorytmu selekcji krawędzi

W ramach implementacji wybranych algorytmów komputerowych zadecydowano o konieczności opracowania algorytmu selekcji krawędzi. W części badawczej pracy dyplomowej dokonano implementacji tego algorytmu zarówno w językach C++, Python oraz Matlab, aby porównać uzyskane wyniki.

Implementacje rozpoczęto od zadeklarowania zmiennych globalnych reprezentujących liczbę wierzchołków, krawędzi oraz wag. Następnie utworzona została struktura edge, której celem jest reprezentacja krawędzi grafu zawierającej numery wierzchołków oraz ich wagę. Fragment kodu odpowiedzialny za opis wstępnych deklaracji przedstawiony został na rysunku 45.

```

int N, M;      // liczba wierzchołków i krawędzi
double W = 0; // suma wag krawędzi w MST

struct edge
{
    unsigned int v1, v2;
    double weight;
} t;

```

Rysunek 45. Deklaracja zmiennych globalnych oraz struktury Edge
[źródło: opracowanie własne]

Klasa comp_edge przedstawia sposób definicji porównania krawędzi w zbiorze, natomiast typ Edge stanowi zbiór krawędzi uporządkowanych według wagi. Struktura vertex odpowiada za reprezentację wierzchołka grafu, a także przechowuje informacje o jego stopniu oraz sąsiadach. Fragment kodu odpowiedzialny za opis struktur danych przedstawiony został na rysunku 46.

```

class comp_edge
{
public:
    bool operator()(const edge& e1, const edge& e2) const
    {
        return e1.weight < e2.weight;
    }
};

typedef multiset<edge, comp_edge> Edge;
Edge edges;
Edge::iterator pos;

struct vertex
{
    unsigned int row, v1, v2;
};

vertex* vs;

```

Rysunek 46. Fragment kodu przedstawiający klasę comp_edge oraz strukturę vertex
[źródło: opracowanie własne]

Analogiczne ten sam fragment kodu postanowiono zaimplementować w języku Python. Fragment kodu odpowiedzialny za deklaracje globalne oraz struktury przedstawiony został na rysunku 47.

```

class Edge:
    def __init__(self, v1: int, v2: int, weight: float):
        self.v1 = v1
        self.v2 = v2
        self.weight = weight

    def __lt__(self, other):
        return self.weight < other.weight

class Vertex:
    def __init__(self):
        self.row = 0
        self.v1 = 0
        self.v2 = 0

```

Rysunek 47. Przedstawienie deklaracji oraz struktur w języku Python
[źródło: opracowanie własne]

W Matlabie część kodu odpowiedzialna za deklarację i przedstawienie struktur wygląda nieco inaczej, ponieważ struktury są reprezentowane za pomocą macierzy. Struktura vertex reprezentowana w kodzie przez macierz vs posiada kolumny, w których każda odpowiada za atrybuty wierzchołka tj. stopień, sąsiada pierwszego oraz drugiego. Z kolei w macierzy edges znajdują się wszystkie krawędzie grafu. Każdy z wierszy macierzy zawiera trzy wartości tj. numer wierzchołka pierwszego oraz drugiego, a także wagę krawędzi. Krawędzie sortowane są za pomocą wag. Fragment kodu ilustrujący tę różnicę przedstawiony został na rysunku 48.

```

function [mst] = mst()
tic;

[N, M, edges] = loadData('C:\\\\Users\\\\mcmys\\\\OneDrive\\\\Pulpit\\\\magisterka_repo\\\\magisterka-repo\\\\Programy\\\\testy\\\\edges100.in');

W = 0; % suma wag krawędzi w MST
L = 0; % suma długości krawędzi w MST
vs = zeros(M+1, 3); % Struktura wierzchołków: [row, v1, v2]

edges = sortrows(edges, 3);

```

Rysunek 48. Przedstawienie deklaracji oraz macierzy w Matlab
[źródło: opracowanie własne]

Kolejnym etapem implementacji algorytmu selekcji krawędzi jest wczytanie danych wejściowych, na których program będzie dokonywał analizy. W języku C++ funkcja date() jest odpowiedzialna za operację pobrania danych z pliku. Następnie uzyskane informacje wczytywane są do zmiennej M oraz N, odpowiadających kolejno za ilość wierzchołków oraz krawędzi. Dodatkowo każda z krawędzi dodana zostaje do zbioru edges, dzięki czemu w późniejszym etapie istnieje możliwość ich przetwarzania. Fragment kodu odpowiedzialny za omawianą operację przedstawiony został na rysunku 49.

```

void date() {
    ifstream dane("C:\\\\Users\\\\mcmys\\\\OneDrive\\\\Pulpit\\\\magisterka_repo\\\\magisterka-repo\\\\Programy\\\\testy\\\\edges100.in");
    dane >> M >> N;
    for (int i = 0; i < N; ++i) {
        dane >> t.v1 >> t.v2 >> t.weight;
        edges.insert(t);
        total_weight += t.weight;
    }
    vs = new vertex[M + 1];
    vertex temp;
    temp.row = 0, temp.v1 = 0, temp.v2 = 0;
    for (int i = 0; i <= M; ++i)
        vs[i] = temp;
    cout << M << ' ' << N << endl;
    dane.close();
}

```

Rysunek 49. Fragment kodu języka C++ odpowiedzialny za wczytanie danych
[źródło: opracowanie własne]

Analogicznie w języku Python funkcja `read_data` odpowiada za wczytanie danych wejściowych. Wewnątrz funkcji następuje otwarcie pliku w trybie odczytu. Pierwsza linia dokumentu po odczytaniu jest parsowana, aby uzyskać wartości `M` oraz `N`. Kolejne linie odczytywane są w celu zebrania informacji o krawędziach tj. numerach wierzchołków oraz ich wagę. Na koniec funkcja zwraca `M`, `N` oraz listę krawędzi. Fragment kodu odpowiedzialny za funkcję `read_data` przedstawiony został na rysunku 50.

```

def read_data(filename: str) -> Tuple[int, int, List[Edge], float]:
    edges = []
    total_weight = 0.0
    with open(filename, 'r') as file:
        M, N = map(int, file.readline().split())
        for _ in range(N):
            v1, v2, weight = map(float, file.readline().split())
            edges.append(Edge(int(v1), int(v2), weight))
            total_weight += weight
    return M, N, edges, total_weight

```

Rysunek 50. Funkcja odpowiedzialna za wczytanie plików w języku Python
[źródło: opracowanie własne]

W Matlabie wczytanie danych wejściowych następuje za pomocą funkcji `loadData`. Całość działa analogicznie do programów napisanych w języku C++ oraz Python. Różnicą jest przechowywanie danych krawędzi w macierzy `edges`, która zawiera w sobie wartości wierzchołka pierwszego i drugiego oraz wagę krawędzi. Rysunek 51 przedstawia fragment kodu Matlab odpowiedzialny za wczytanie plików.

```

function [N, M, edges, total_weight] = loadData(filename)
    fileID = fopen(filename, 'r');
    data = fscanf(fileID, '%d %d', [1, 2]);
    M = data(1);
    N = data(2);
    edges = fscanf(fileID, '%d %d %f', [3, Inf]);
    fclose(fileID);
    total_weight = sum(edges(:, 3));
end

```

Rysunek 51. Wczytanie pliku w Matlab
[źródło: opracowanie własne]

W kolejnym etapie implementacji algorytmu należy sprawdzić czy dodane krawędzie do minimalnego drzewa rozpinającego nie tworzą cyklu. Do tego celu wykorzystana została funkcja „nocycle()”, która korzysta ze struktury vs w celu śledzenia połączeń pomiędzy wierzchołkami. Wierzchołki iterowane są przez wierzchołki do momentu odnalezienia takich, które nie powodują cyklu, zwracając wartość true lub false w zależności od uzyskanego wyniku sprawdzenia. Funkcja odpowiedzialna za sprawdzenie cyklu w języku C++ przedstawiona została na rysunku 52.

```

bool nocycle(edge e)
{
    unsigned int v1 = e.v1, v2 = e.v2;
    v1 = vs[v1].v1 + vs[v1].v2;
    while (v1 != v2 && vs[v1].row == 2)
    {
        if (vs[v1].v1)
            v1 = vs[v1].v1;
        else
            v1 = vs[v1].v2;
    }
    return v1 != v2;
}

```

Rysunek 52. Funkcja nocycle w języku C++
[źródło: opracowanie własne]

Analogicznie wygląda proces sprawdzenia występowania cyklu w języku Python. Na początku pobierane są numery wierzchołków z krawędzi. W pętli while zachodzi iteracja przez wierzchołki, sprawdzając czy występuje cykl. W przypadku gdy liczba sąsiadów wierzchołka wyniesie 2, pętla sprawdza dostępność sąsiadów oraz dokonuje aktualizacji wartości „v1”, która odpowiada sumie sąsiadów. Jeżeli cykl nie występuje, funkcja zwraca

wartość „true”, w przeciwnym razie zwracana jest wartość „False”. Fragment kodu w języku Python odpowiedzialny za badanie występowania cyklu przedstawiony został na rysunku 53.

```
def nocycle(vs: List[Vertex], e: Edge) -> bool:
    v1, v2 = e.v1, e.v2
    v1 = vs[v1].v1 + vs[v1].v2
    while v1 != v2 and vs[v1].row == 2:
        if vs[v1].v1:
            v1 = vs[v1].v1
        else:
            v1 = vs[v1].v2
    return v1 != v2
```

Rysunek 53. Przedstawienie funkcji nocycle() w języku Python
[źródło: opracowanie własne]

W przypadku języka Matlab za sprawdzenie występowania cyklu odpowiadają dwie funkcje nocycle oraz findroot. Pierwsza z nich przyjmuje macierz wierzchołków oraz argument e, który odnosi się do krawędzi reprezentowanej przez wektor. Funkcja wyodrębnia numery wierzchołków oraz wywołuje funkcję findroot w celu odnalezienia korzenia dla tych wierzchołków. W przypadku, gdy korzenie są różne, wybrana krawędź może zostać dodana do minimalnego drzewa rozpinającego bez tworzenia cyklu. Fragment kodu odpowiedzialny za ten proces przedstawiony został na rysunku 54.

```
function cycle = nocycle(vs, e)
    v1 = e(1);
    v2 = e(2);
    v1_root = findroot(vs, v1);
    v2_root = findroot(vs, v2);
    cycle = (v1_root ~= v2_root);
end

function root = findroot(vs, v)
    while vs(v, 1) == 2
        if vs(v, 2) ~= 0
            v = vs(v, 2);
        else
            v = vs(v, 3);
        end
    end
    root = v;
end
```

Rysunek 54. Funkcje sprawdzające występowanie cyklu w Matlab
[źródło: opracowanie własne]

Kolejnym etapem implementacji była budowa minimalnego drzewa rozpinającego. Do tego celu wykorzystana została funkcja build(). Proces dodawania krawędzi do drzewa przebiega zgodnie z ich wagą, aby uniknąć powstania cyklu. Fragment kodu odpowiedzialny za budowę MST w języku C++ przedstawiony został na rysunku 55 umieszczonym poniżej.

```
void build()
{
    pos = edges.begin();
    for (int i = 1; i <= M; ++i)
    {
        t = *pos;
        if (vs[t.v1].row == 2 || vs[t.v2].row == 2)
        {
            ++pos;
            --i;
            continue;
        }
        if (vs[t.v1].row + vs[t.v2].row <= 1)
        {
            if (vs[t.v1].v1)
                vs[t.v1].v1 = t.v1;
            else
                vs[t.v1].v2 = t.v1;
            if (vs[t.v2].v1)
                vs[t.v2].v1 = t.v2;
            else
                vs[t.v1].v2 = t.v2;
            ++vs[t.v1].row, ++vs[t.v2].row, ++pos;
            W += t.weight;
            continue;
        }
        if (vs[t.v1].row + vs[t.v2].row == 2)
        {
            if (nocycle(t) || i == M)
            {
                cout << '\n'
                    << t.weight << ' ' << t.v1 << ' ' << t.v2 << " II " << i;
                ++vs[t.v1].row, ++vs[t.v2].row, ++pos;
                W += t.weight;
            }
            else
            {
                ++pos;
                --i;
            }
            continue;
        }
    }
}
```

Rysunek 55. Funkcja budująca MST w języku C++.
[źródło: opracowanie własne]

W analogiczny sposób zaimplementowana została funkcja budująca minimalne drzewo rozpinające w języku Python. Fragment kodu przedstawiony został na rysunku 56.

```

def build(M: int, edges: List[Edge]) -> Tuple[float, float]:
    vs = [Vertex() for _ in range(M + 1)]
    edges.sort()
    W = 0.0
    L = 0.0

    i = 1
    for t in edges:
        if vs[t.v1].row == 2 or vs[t.v2].row == 2:
            continue
        if vs[t.v1].row + vs[t.v2].row <= 1:
            if vs[t.v1].v1:
                vs[t.v1].v1 = t.v1
            else:
                vs[t.v1].v2 = t.v1
            if vs[t.v2].v1:
                vs[t.v2].v1 = t.v2
            else:
                vs[t.v2].v2 = t.v2
            vs[t.v1].row += 1
            vs[t.v2].row += 1
            W += t.weight
            L += fabs(t.v1 - t.v2)
            i += 1
            if i > M:
                break
        elif vs[t.v1].row + vs[t.v2].row == 2:
            if nocycle(vs, t) or i == M:
                print(f'\n{t.weight} {t.v1} {t.v2} II {i}')
                vs[t.v1].row += 1
                vs[t.v2].row += 1
                W += t.weight
                L += fabs(t.v1 - t.v2)
                i += 1
                if i > M:
                    break

    return W, L

```

Rysunek 56. Funkcja budująca MST w języku Python.
[źródło: opracowanie własne]

W Matlabie główna pętla iteruje przez posortowane krawędzie, sprawdzając, czy mogą one zostać dodane do MST bez tworzenia cyklu. Jeśli krawędź nie tworzy cyklu, jest ona dodawana do MST, a odpowiednie zmienne są aktualizowane. W ten sposób program efektywnie buduje MST, minimalizując tym samym wagę drzewa. Fragment kodu odpowiedzialny za budowę minimalnego drzewa rozpinającego w Matlabie przedstawiony został na rysunku 57.

```

edge_count = 0;
i = 1;
while edge_count < M
    t = edges(i, :);
    if vs(t(1), 1) < 2 && vs(t(2), 1) < 2
        if nocycle(vs, t)
            disp([num2str(t(3)) ' ' num2str(t(1)) ' ' num2str(t(2)) ' II ' num2str(edge_count + 1)]);
            vs(t(1), 1) = vs(t(1), 1) + 1;
            vs(t(2), 1) = vs(t(2), 1) + 1;
            if vs(t(1), 2) == 0
                vs(t(1), 2) = t(2);
            else
                vs(t(1), 3) = t(2);
            end
            if vs(t(2), 2) == 0
                vs(t(2), 2) = t(1);
            else
                vs(t(2), 3) = t(1);
            end
            W = W + t(3);
            L = L + abs(t(1) - t(2));
            edge_count = edge_count + 1;
        end
    end
    i = i + 1;
end

```

Rysunek 57. Fragment kodu odpowiedzialny za budowę MST w Matlabie.
[źródło: opracowanie własne]

Kolejnym elementem implementacji algorytmu selekcji krawędzi był zapis informacji wyświetlonych w programie tj. sumy wag krawędzi, czasu wykonania programu oraz ilości wierzchołków do pliku CSV. Fragment kodu odpowiedzialny za omawianą procedurę został przedstawiony na rysunku 58.

```
void save_to_csv(double weight_sum, double duration, int vertices, double total_weight) {
    ofstream csv_file;
    csv_file.open("result.csv", ios::out | ios::app);
    if (csv_file.is_open()) {
        if (csv_file.tellp() == 0) {
            csv_file << "Weight Sum,Execution Time (s),Vertices,Total Weight" << endl;
        }
        csv_file << fixed << setprecision(2) << weight_sum << "," << duration << "," << vertices << "," << total_weight << endl;
    }
    else {
        cout << "Unable to open file to write CSV data." << endl;
    }
}
```

Rysunek 58. Funkcja zapisująca dane do pliku CSV w języku C++.

[źródło: opracowanie własne]

W sposób analogiczny zaimplementowano funkcję zapisu danych z programu do pliku CSV w języku Python. Fragment kodu odpowiedzialny za tą operację przedstawiony został na rysunku 59.

```
def save_to_csv(weight_sum: float, duration: float, vertices: int, total_weight: float):
    with open('result.csv', mode='a', newline='') as file:
        writer = csv.writer(file)
        file_is_empty = file.tell() == 0
        if file_is_empty:
            writer.writerow(["Weight Sum", "Execution Time (s)", "Vertices", "Total Weight"])
        writer.writerow([f'{weight_sum:.2f}', f'{duration:.6f}', vertices, f'{total_weight:.2f}'])
```

Rysunek 59. Funkcja zapisująca dane do pliku CSV w języku Python.

[źródło: opracowanie własne]

W równie podobny sposób funkcja zapisu danych do pliku CSV została zaimplementowana w Matlabie. Fragment kodu przeprowadzający omawianą operację przedstawiony został na rysunku 60.

```
function save_to_csv(weight_sum, duration, vertices, total_weight)
    filename = 'result.csv';
    fileID = fopen(filename, 'a');
    if fileID == -1
        error('Unable to open file to write CSV data.');
    end
    if fseek(fileID) == 0
        fprintf(fileID, 'Weight Sum,Execution Time (s),Vertices,Total Weight\n');
    end
    fprintf(fileID, '%.2f,.6f,%d,.2f\n', weight_sum, duration, vertices, total_weight);
    fclose(fileID);
end
```

Rysunek 60. Funkcja zapisująca dane do pliku CSV w Matlabie.

[źródło: opracowanie własne]

Ostatnim elementem implementacji algorytmu była funkcja main(). Główna część programu przeznaczona została do wyliczania czasu działania programu oraz informowaniu użytkownika o otrzymanych rezultatach tj. sumie wag krawędzi w MST oraz czasie działania programu. Fragment kodu przedstawiający omówioną implementację w języku C++ przedstawiony został na rysunku 61.

```
int main() {
    auto start = high_resolution_clock::now();

    date();
    build();

    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);

    double duration_seconds = duration.count() / 1000.0;

    cout << fixed << setprecision(2) << '\n' << "Sum of edge weights in MST: " << W << endl;
    cout << "Total weight of the path: " << total_weight << endl;
    cout << "Execution time: " << duration_seconds << " s" << endl;

    save_to_csv(W, duration_seconds, M, total_weight);

    cin.get();
    return 0;
}
```

Rysunek 61. Implementacja funkcji main() w języku C++.
[źródło: opracowanie własne]

W analogiczny sposób funkcja główna programu została zaimplementowana w języku Python. Różnicą pomiędzy kodem C++, a Pythonem jest wczytanie pliku z danymi wejściowymi, który w Pythonie wykonuje się w funkcji main(). Fragment implementacji przedstawiony został na rysunku 62.

```
def main():
    start = time.time()

    M, N, edges, total_weight = read_data('C:\\Users\\mcmys\\OneDrive\\Pulpit\\magisterka_repo\\magisterka-repo\\Programy\\1'
    W = build(M, edges)

    end = time.time()
    duration = end - start

    print(f'\nSuma wag krawędzi w MST: {W:.2f}')
    print(f'Total weight of the path: {total_weight:.2f}')
    print(f'Czas działania programu: {duration:.6f} s')

    save_to_csv(W, duration, M, total_weight)
```

Rysunek 62. Implementacja funkcji main() w języku Python.
[źródło: opracowanie własne]

W Matlabie funkcja mst pełni rolę głównej funkcji programu. Wykorzystana została do inicjalizacji całego procesu, uruchomienia pomiaru czasu, wczytania danych, zbudowania minimalnego drzewa rozpinającego, wyświetlenia wyników oraz zapisania ich do pliku CSV. Na rysunku 63 przedstawiona została implementacja funkcji mst w Matlabie.

```

function mst()
tic;

[N, M, edges, total_weight] = loadData('C:\Users\mcmys\OneDrive\Pulpit\magisterka_repo\magisterka-repo\Programy\testy\edges100.in');

W = 0; % suma wag krawędzi w MST
vs = zeros(M+1, 3); % Struktura wierzchołków: [row, v1, v2]

edges = sortrows(edges, 3);

edge_count = 0;
i = 1;
while edge_count < M
    t = edges(i, :);
    if vs(t(1), 1) < 2 && vs(t(2), 1) < 2
        if nocycle(vs, t)
            disp(['Suma wag krawędzi w MST: ' num2str(W, '.2f')]);
            disp(['Total weight of the path: ' num2str(total_weight, '.2f')]);
            disp(['Czas działania programu: ' num2str(execution_time, '.6f') ' s']);
            save_to_csv(W, execution_time, M, total_weight);
        end
        i = i + 1;
    end
end

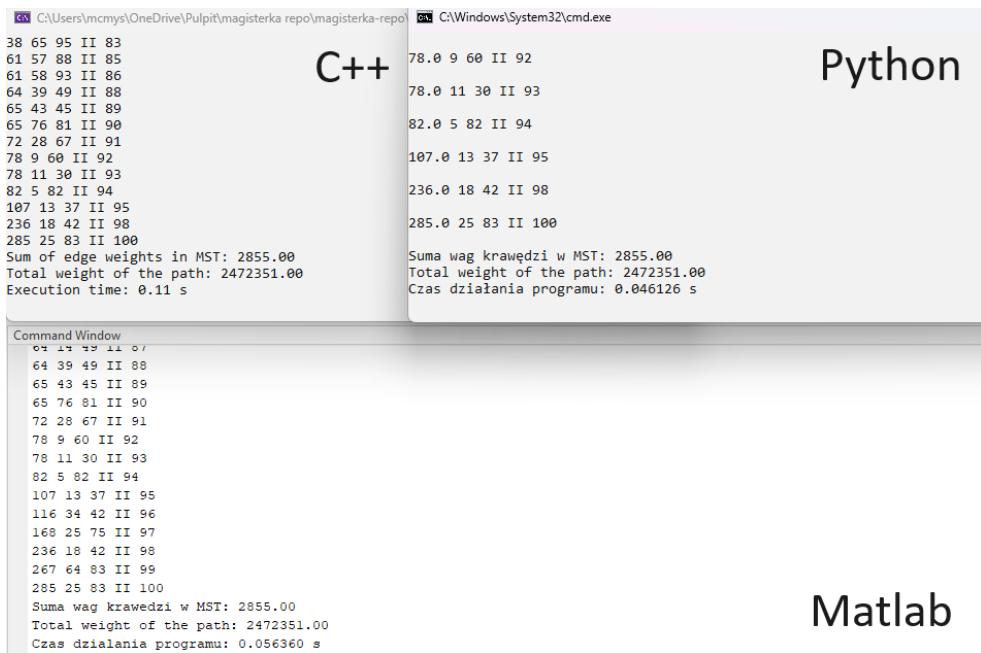
execution_time = toc;

disp(['Suma wag krawędzi w MST: ' num2str(W, '.2f')]);
disp(['Total weight of the path: ' num2str(total_weight, '.2f')]);
disp(['Czas działania programu: ' num2str(execution_time, '.6f') ' s']);


```

Rysunek 63. Funkcja mst() w Matlab
[źródło: opracowanie własne]

W celu potwierdzenia poprawności działania opracowanych algorytmów selekcji krawędzi w języku C++, Python oraz Matlab przeprowadzono badanie na wygenerowanych podczas przedstawiania działania generatora liczb losowych danych wejściowych. Efekt działania przedstawiony został na rysunku 64.



Platform	Output
C++	<pre> 38 65 95 II 83 61 57 88 II 85 61 58 93 II 86 64 39 49 II 88 65 43 45 II 89 65 76 81 II 90 72 28 67 II 91 78 9 60 II 92 78 11 30 II 93 82 5 82 II 94 107 13 37 II 95 236 18 42 II 98 285 25 83 II 100 Sum of edge weights in MST: 2855.00 Total weight of the path: 2472351.00 Execution time: 0.11 s </pre>
Python	<pre> 78.0 9 60 II 92 78.0 11 30 II 93 82.0 5 82 II 94 107.0 13 37 II 95 236.0 18 42 II 98 285.0 25 83 II 100 Suma wag krawędzi w MST: 2855.00 Total weight of the path: 2472351.00 Czas działania programu: 0.046126 s </pre>
Matlab	<pre> Command Window 64 14 99 II 87 64 39 49 II 88 65 43 45 II 89 65 76 81 II 90 72 28 67 II 91 78 9 60 II 92 78 11 30 II 93 82 5 82 II 94 107 13 37 II 95 116 34 42 II 96 168 25 75 II 97 236 18 42 II 98 267 64 83 II 99 285 25 83 II 100 Suma wag krawędzi w MST: 2855.00 Total weight of the path: 2472351.00 Czas działania programu: 0.056360 s </pre>

Rysunek 64. Efekt działania programu w C++, Pythonie oraz Matlabie.
[źródło: opracowanie własne]

Rysunek 64 przedstawia wyniki uzyskane w wyniku przeprowadzenia badania na danych wejściowych dla 100 wierzchołków. Programy zgodnie uzyskały wynik sum wag krawędzi w minimalnym drzewie rozpinającym równy 2855. Jedyną różnicą pomiędzy programami jest czas działania programu. Najkrótszy czas uzyskany został w algorytmie zaimplementowanym w języku Python. Na pozycji drugiej znajduje się wynik uzyskany przez kod w Matlabie. Ostatnie miejsce z czasem 0.11 sekundy uzyskał program opracowany w języku C++.

6.3. Implementacja algorytmu dwu-optymalnego.

W ramach implementacji wybranych algorytmów komputerowych zadecydowano o konieczności opracowania algorytmu dwu-optymalnego. W części badawczej pracy dyplomowej dokonano implementacji tego algorytmu zarówno w językach C++, Python oraz Matlab, aby porównać uzyskane wyniki.

Implementacja rozpoczęta została od deklaracji zmiennych globalnych, które wykorzystane zostały do przechowywania danych wejściowych takich jak liczba wierzchołków oraz krawędzi, przechowywania wyników, a także do implementacji logiki algorytmu tj. zmienne iteracyjne, pomocnicze oraz przechowujące stany tymczasowe. Do tych wartości należą:

- N – określa liczbę wierzchołków w grafie;
- M - określa liczbę krawędzi w grafie;
- Ptr – tablica wskaźników używana do przechowywania trasy w algorytmie;
- W - dwuwymiarowa tablica dynamiczna (macierz) przechowująca wagi krawędzi między wierzchołkami;
- Route - tablica przechowująca aktualną trasę odwiedzanych wierzchołków;
- Ahead - pomocnicza zmienna wskaźnikowa używana w przestawianiu trasy;
- I, I1, I2 - zmienne iteracyjne i pomocnicze używane w pętlach oraz w obliczeniach trasy;
- Index - zmienna przechowująca indeks wierzchołka, używana do aktualizacji trasy;
- J, J1, J2 - zmienne iteracyjne i pomocnicze używane w pętlach oraz w obliczeniach trasy;
- Last - pomocnicza zmienna wskaźnikowa używana w przestawianiu trasy;
- Limit - zmienna określająca limit iteracji w pętli;

- Max - zmienna przechowująca maksymalną różnicę wag krawędzi, używana do optymalizacji trasy;
- Max1 - tymczasowa zmienna przechowująca aktualną różnicę wag krawędzi w pętli;
- Next - pomocnicza zmienna wskaźnikowa używana w przedstawianiu trasy;
- S1, S2 - zmienne przechowujące wskaźniki na wierzchołki początkowe krawędzi w przedstawianiu trasy;
- T1, T2 - zmienne przechowujące wskaźniki na wierzchołki końcowe krawędzi w przedstawianiu trasy;
- Tweight - całkowita waga trasy, która jest minimalizowana przez algorytm;
- TotalWeight - suma wag wszystkich krawędzi w grafie, używana do celów informacyjnych i weryfikacyjnych.

Fragment kodu obrazujący deklarację zmiennych przedstawiony został na rysunku 65.

```
int N, M;
int* Ptr;
int** W;
int* Route;
int Ahead, I, I1, I2, Index, J, J1, J2, Last, Limit, Max, Max1, Next, S1, S2, T1, T2;
int Tweight = 0;
int TotalWeight = 0;
```

Rysunek 65. Deklaracja zmiennych globalnych w języku C++.
[źródło: opracowanie własne]

W sposób analogiczny zadeklarowane zostały zmienne globalne w języku Python, co zostało przedstawione na rysunku 66.

```
N = 0
M = 0
Ptr = []
W = []
Route = []
Ahead = I = I1 = I2 = Index = J = J1 = J2 = Last = Limit = Max = Max1 = Next = S1 = S2 = T1 = T2 = 0
Tweight = 0
TotalWeight = 0
```

Rysunek 66. Deklaracja zmiennych globalnych w języku Python.
[źródło: opracowanie własne]

Z kolei w kodzie Matlab, zmienne rozproszone zostały po różnych funkcjach tj. Initialization, calculateInitialWeight oraz Calculations. Z tego powodu postanowiono

przedstawić inicjalizację przykładowej zmiennej, a dokładniej calculateInitialWeight, zawierającej deklarację zmiennej Tweight, co zostało zaprezentowane na rysunku 67.

```
function Tweight = calculateInitialWeight(N, W, Route)
    Tweight = 0;
    for i = 1:N-1
        Tweight = Tweight + W(Route(i), Route(i + 1));
    end
    Tweight = Tweight + W(Route(N), Route(1));
end
```

Rysunek 67. Deklaracja zmiennej calculateInitialWeight w Matlab.
[źródło: opracowanie własne]

W kolejnym etapie implementacji przedstawiona została funkcja inicjalizacji danych, której zadaniem było wczytanie danych z pliku, zainicjalizowanie tablico wraz z zmiennymi oraz obliczenie początkowej wartości wagi trasy. Wykorzystywana została również do wczytania informacji dotyczących krawędzi grafu oraz w celu zsumowania ich wag. Fragment kodu odpowiedzialny za ten proces w języku C++ przedstawiony został na rysunku 68.

```
void Initialization()
{
    int X, Y, Temp;
    ifstream date("C:\\Users\\mcmys\\OneDrive\\Pulpit\\magisterka_repo\\magisterka-repo\\Programy\\testy\\edges100.in");
    date >> N >> M;
    Ptr = new int[N + 1];
    Route = new int[N + 1];
    W = new int* [N + 1];
    for (int i = 0; i <= N; ++i) W[i] = new int[N + 1];
    for (int i = 1; i <= M; ++i)
    {
        date >> X >> Y >> Temp;
        W[X][Y] = Temp;
        W[Y][X] = Temp;
        TotalWeight += Temp;
    }
    for (int i = 1; i <= N; ++i) Route[i] = i, W[i][i] = 0;
    for (int i = 1; i < N; ++i) Tweight += W[i][i + 1];
    Tweight += W[N][1];
}
```

Rysunek 68. Przedstawienie funkcji Initialization() w języku C++.
[źródło: opracowanie własne]

W sposób analogiczny zmienna Initialization() zaimplementowana została w języku Python. Na początku funkcja przeprowadza operacje na pliku zawierającym dane wejściowe tj. otwiera plik, dokonuje odczytu danych oraz zamyka źródło danych. Następnie pierwsze dwie liczby pliku przypisane zostały jako liczba wierzchołków oraz krawędzi. W kolejnym etapie dokonana została inicjalizacja listy Ptr oraz Route wraz z macierzą „W” jako tablica numpy, której rozmiar wyniósł $(N + 1) \cdot (N + 1)$ z wartościami zerowymi. Następnie funkcja odczytuje krawędzie z danych wejściowych i dokonuje zapisu ich wag do macierzy „W”.

Następnie dokonał zsumowania wag wszystkich krawędzi, ustawił wartość na przekątnej macierzy „W” na zero oraz wykonał obliczenie początkowej trasy. Fragment kodu odpowiedzialny za ten proces przedstawiony został na rysunku 69.

```
def initialization():
    global N, M, Ptr, Route, W, Tweight, TotalWeight
    with open("C:\\\\Users\\\\mcmys\\\\OneDrive\\\\pulpit\\\\magisterka_repo\\\\magisterka-repo\\\\Programy\\\\testy\\\\edges100.in", "r") as
        N, M = map(int, file.readline().split())
        Ptr = [0] * (N + 1)
        Route = [0] * (N + 1)
        W = np.zeros((N + 1, N + 1), dtype=int)

    for _ in range(M):
        X, Y, Temp = map(int, file.readline().split())
        W[X][Y] = Temp
        W[Y][X] = Temp
        TotalWeight += Temp

    for i in range(1, N + 1):
        Route[i] = i
        W[i][i] = 0
    for i in range(1, N):
        Tweight += W[i][i + 1]
    Tweight += W[N][1]
```

Rysunek 69. Przedstawienie funkcji Initialization() w języku Python.
[źródło: opracowanie własne]

Fragment kodu inicjalizacji przedstawiony został również dla Matlaba, za pomocą którego algorytm został zaimplementowany. Na początku funkcja dokonuje otwarcia pliku, wczytania danych oraz zamknięcia źródła danych. Następnie pierwsze dwie liczby z pliku są przypisywane do N oraz M, odpowiadających kolejno za liczbę wierzchołków oraz krawędzi. Następnie zainicjalizowana została macierz W o rozmiarze N x N z wartościami zerowymi. W kolejnym kroku dokonano inicjalizacji trasy Route jako sekwencję liczb od 1 do N. Etap kolejny to wczytanie krawędzi z pliku źródłowego oraz ich zapis do macierzy W, aby kolejno dokonać sumowania wag wszystkich krawędzi do TotalWeight. Na koniec ustaliona została wartość na przekątnej macierzy W jako zero. Fragment kodu odpowiedzialny za ten proces przedstawiony został na rysunku 70.

```

function [N, M, W, Route, TotalWeight] = Initialization()
    filename = 'C:\Users\mcmys\OneDrive\Pulpit\magisterka repo\magisterka-repo\Programy\testy\edges100.in';
    fileID = fopen(filename, 'r');
    data = fscanf(fileID, '%d');
    fclose(fileID);

    N = data(1);
    M = data(2);
    W = zeros(N, N);
    Route = 1:N;
    TotalWeight = 0;

    idx = 3;
    for i = 1:M
        X = data(idx);
        Y = data(idx + 1);
        Temp = data(idx + 2);
        W(X, Y) = Temp;
        W(Y, X) = Temp;
        TotalWeight = TotalWeight + Temp;
        idx = idx + 3;
    end

    for i = 1:N
        W(i, i) = 0;
    end
end

```

Rysunek 70. Przedstawienie funkcji Initialization() w języku Matlabie.

[źródło: opracowanie własne]

W kolejnym etapie implementacji przedstawiona została funkcja obliczeniowa Calculations(), w której zoptymalizowano trasę w grafie za pomocą algorytmu dwu-optimalnego. Na początku zainicjalizowana została tablica wskaźników Ptr, w celu utworzenia trasy cyklicznej. W pętli do-while dokonano przeszukania wszystkich możliwych par krawędzi, aby odnaleźć takie, których zamiana skróciłaby długość trasy. Po znalezieniu lepszej trasy, zamieniono krawędzie i zaktualizowano wskaźniki Ptr. Proces powtarzany był do momentu, w którym nie odnaleziono dalszych ulepszeń. W fazie końcowej dokonano aktualizacji trasy zgodnie z nowymi wskaźnikami, co skutkowało optymalną trasą w tablicy Route i zredukowaną wagą Tweight. Proces ten przedstawiony został na rysunku 71.

```

void Calculations()
{
    for (I = 1; I < N; ++I) Ptr[Route[I]] = Route[I + 1];
    Ptr[Route[N]] = Route[1];
    do
    {
        Max = 0, I1 = 1;
        for (int I = 1; I <= N - 2; ++I)
        {
            if (I == 1) Limit = N - 1; else Limit = N;
            I2 = Ptr[I1], J1 = Ptr[I2];
            for (J = I + 2; J <= Limit; ++J)
            {
                J2 = Ptr[J1], Max1 = W[I1][I2] + W[J1][J2] - (W[I1][J1] + W[I2][J2]);
                if (Max1 > Max) S1 = I1, S2 = I2, T1 = J1, T2 = J2, Max = Max1;
                J1 = J2;
            }
            I1 = I2;
        }
        if (Max > 0)
        {
            Ptr[S1] = T1, Next = S2, Last = T2;
            do Ahead = Ptr[Next], Ptr[Next] = Last, Last = Next, Next = Ahead; while (Next != T2);
            Tweight -= Max;
        }
    } while (Max != 0);
    Index = 1;
    for (I = 1; I <= N; ++I) Route[I] = Index, Index = Ptr[Index];
}

```

Rysunek 71. Przedstawienie funkcji Calculations() w języku C++.

[źródło: opracowanie własne]

W sposób analogiczny dokonano implementacji funkcji obliczeniowej w języku Python. Na początku zainicjalizowano tablicę wskaźników Ptr, w celu utworzenia cyklicznej trasy. W głównej pętli while dokonało się przeszukanie wszystkich par krawędzi, w celu odnalezienia tych, których zamiana skróci długość trasy. W momencie odnalezienia zmiany wartości „S1”, „S2”, „T1” oraz „T2” zostały zaktualizowane, natomiast krawędzie zamienione. Proces powtarzał się do momentu, w którym nie było możliwości odnaleźć dalszych ulepszeń, co w konsekwencji zakończyło pętle. Proces ten został przedstawiony na rysunku 72.

```
def calculations():
    global Ptr, Route, Tweight, Max, I1, S1, S2, T1, T2, Max1
    for I in range(1, N):
        Ptr[Route[I]] = Route[I + 1]
    Ptr[Route[N]] = Route[1]
    while True:
        Max = 0
        I1 = 1
        for I in range(1, N - 1):
            Limit = N if I != 1 else N - 1
            I2 = Ptr[I1]
            J1 = Ptr[I2]
            for J in range(I + 2, Limit + 1):
                J2 = Ptr[J1]
                Max1 = W[I1][I2] + W[J1][J2] - (W[I1][J1] + W[I2][J2])
                if Max1 > Max:
                    S1, S2, T1, T2 = I1, I2, J1, J2
                    Max = Max1
                J1 = J2
            I1 = I2
        if Max <= 0:
            break
        Ptr[S1] = T1
        Next = S2
        Last = T2
        while Next != T2:
            Ahead = Ptr[Next]
            Ptr[Next] = Last
            Last = Next
            Next = Ahead
            Tweight -= Max

        Index = 1
        for I in range(1, N + 1):
            Route[I] = Index
        Index = Ptr[Index]
```

Rysunek 72. Przedstawienie funkcji Calculations() w języku Python.
[źródło: opracowanie własne]

Fragment kodu inicjalizacji funkcję obliczeniową przedstawiony został również w Matlabie. Najpierw zainicjalizowano tablicę wskaźników Ptr, aby utworzyć cykliczną trasę. Następnie w pętli while przeszukano wszystkie możliwe pary krawędzi w celu znalezienia tych, których zamiana skróciłaby długość trasy. Po znalezieniu lepszej trasy zamieniono krawędzie i zaktualizowano wskaźniki Ptr, kontynuując proces aż do braku dalszych ulepszeń. Na koniec zaktualizowano tablicę Route zgodnie z nowymi wskaźnikami, co skutkowało optymalną trasą i zmniejszoną wagą Tweight. Kod Matlab, w którym zainicjalizowano funkcję obliczeniową został przedstawiony na rysunku 73.

```

function [Ptr, Route, Tweight] = Calculations(N, W, Route, Tweight)
    Ptr = zeros(1, N);
    for I = 1:N-1
        Ptr(Route(I)) = Route(I + 1);
    end
    Ptr(Route(N)) = Route(1);

    while true
        Max = 0;
        I1 = 1;
        for I = 1:N-2
            if I == 1
                Limit = N - 1;
            else
                Limit = N;
            end
            I2 = Ptr(I1);
            J1 = Ptr(I2);
            for J = I + 2:Limit
                J2 = Ptr(J1);
                Max1 = W(I1, I2) + W(J1, J2) - (W(I1, J1) + W(I2, J2));
                if Max1 > Max
                    S1 = I1;
                    S2 = I2;
                    T1 = J1;
                    T2 = J2;
                    Max = Max1;
                end
                J1 = J2;
            end
            I1 = I2;
        end
        if Max > 0
            Ptr(S1) = T1;
            Next = S2;
            Last = T2;
            while true
                Ahead = Ptr(Next);
                Ptr(Next) = Last;
                Last = Next;
                Next = Ahead;
                if Next == T2
                    break;
                end
            end
            Tweight = Tweight - Max;
        else
            break;
        end
    end

    Index = 1;
    for I = 1:N
        Route(I) = Index;
        Index = Ptr(Index);
    end
end

```

Rysunek 73. Funkcja obliczeniowa Calculations przedstawiona w Matlabie.
[źródło: opracowanie własne]

Ostatnia faza implementacji, to przedstawienie funkcji głównej programu. Na początku zmierzono czas wykonania programu za pomocą funkcji `high_resolution_clock`. W kolejnym etapie wywołano funkcję `Initialization`, aby zainicjalizować dane wejściowe. Wyświetlono fragment macierzy wag dla weryfikacji. Po wykonaniu obliczeń w funkcji `Calculations` wyświetlono całkowitą wagę trasy oraz trasę. Podano również sumę wszystkich wag krawędzi w grafie. Zmierzony czas wykonania programu wyświetlono w sekundach z dokładnością do trzech miejsc po przecinku. Wyniki zapisano do pliku CSV, a program zakończono po wcisnięciu klawisza. Proces ten został przedstawiony na rysunku 74.

```

int main()
{
    auto start = high_resolution_clock::now();

    Initialization();

    cout << "Displaying a portion of the weight matrix for verification:" << endl;
    for (int k = 1; k <= min(10, N); ++k)
    {
        for (int l = 1; l <= min(10, N); ++l)
        {
            cout << W[k][l] << '\t';
        }
        cout << '\n';
    }

    Calculations();

    cout << "Total weight of the route: " << Tweight << endl;
    cout << "Route: ";
    for (int i = 1; i <= N; ++i) cout << Route[i] << " ";
    cout << endl;

    cout << "Sum of all edge weights in the graph: " << TotalWeight << endl;

    auto end = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end - start).count();
    double seconds = duration / 1e6;
    cout << "Execution time: " << fixed << setprecision(3) << seconds << " s" << endl;

    ofstream outputFile("output.csv");
    outputFile << "Total weight of the route," << Tweight << "\n";
    outputFile << "Route,";
    for (int i = 1; i <= N; ++i) outputFile << Route[i] << (i < N ? "," : "\n");
    outputFile << "Sum of all edge weights in the graph," << TotalWeight << "\n";
    outputFile << "Execution time," << fixed << setprecision(3) << seconds << " s\n";
    outputFile.close();

    cin.get();
    return 0;
}

```

Rysunek 74. Funkcja main() w języku C++.
[źródło: opracowanie własne]

W sposób analogiczny dokonano implementacji funkcji main w języku Python. Na początku zmierzono czas wykonania programu za pomocą funkcji time.time(). Następnie wywołano funkcję initialization, aby zainicjalizować dane wejściowe. Wyświetlono fragment macierzy wag dla weryfikacji. Po wykonaniu obliczeń w funkcji calculations wyświetlono całkowitą wagę trasy oraz trasę. Obliczono i wyświetlono sumę wszystkich długości krawędzi w trasie oraz całkowitą wagę krawędzi w grafie. Zmierzony czas wykonania programu wyświetlono w sekundach. Fragment implementacji funkcji main() w języku Python przedstawiono na rysunku 75.

```

def main():
    start = time.time()

    initialization()
    print("Displaying a portion of the weight matrix for verification:")
    for k in range(1, min(10, N + 1)):
        print('\t'.join(map(str, W[k][1:min(10, N + 1)])))
    calculations()
    print("Total weight of the route:", Tweight)
    print("Route:", ' '.join(map(str, Route[1:])))

    EdgeSum = 0
    for i in range(1, N):
        EdgeSum += W[Route[i]][Route[i + 1]]
    EdgeSum += W[Route[N]][Route[1]]

    print("Sum of all edge lengths in the route:", EdgeSum)
    print("Sum of all edge weights in the graph:", TotalWeight)

    end = time.time()
    duration = end - start
    print("Execution time:", duration, "seconds")

```

Rysunek 75. Funkcja main() w języku Python.
[źródło: opracowanie własne]

Tą samą funkcję zaimplementowano również w Matlabie. Rozpoczęto od zmierzenia czasu wykonania programu za pomocą funkcji tic i toc. Następnie wywołano funkcję Initialization, aby zainicjalizować dane wejściowe, takie jak liczba wierzchołków N, liczba krawędzi M, macierz wag W, trasa Route oraz całkowita waga krawędzi TotalWeight. Początkową wagę trasy Tweight obliczono za pomocą funkcji calculateInitialWeight. Następnie wywołano funkcję Calculations, która zoptymalizowała trasę, zwracając zaktualizowane tablice Ptr, Route oraz nową wartość Tweight. Wyświetlono fragment macierzy wag dla weryfikacji. Po zakończeniu obliczeń wyświetlono całkowitą wagę trasy oraz trasę. Obliczono i wyświetlono sumę wszystkich długości krawędzi w trasie (EdgeSum) oraz całkowitą wagę wszystkich krawędzi w grafie (TotalWeight). Na koniec zmierzono i wyświetlono czas wykonania programu w sekundach z dokładnością do trzech miejsc po przecinku. Fragment kodu przedstawiający implementację funkcji main() w Matlabie został przedstawiony na rysunku 76.

```

function main
tic;

[N, M, W, Route, TotalWeight] = Initialization();
Tweight = calculateInitialWeight(N, W, Route);

[Ptr, Route, Tweight] = Calculations(N, W, Route, Tweight);

disp('Displaying a portion of the weight matrix for verification:');
displayMatrix(W, min(10, N));

disp('Total weight of the route:');
disp(Tweight);

disp('Route:');
disp(Route);

EdgeSum = 0;
for i = 1:N-1
    EdgeSum = EdgeSum + W(Route(i), Route(i + 1));
end
EdgeSum = EdgeSum + W(Route(N), Route(1));

disp('Sum of all edge lengths in the route:');
disp(EdgeSum);

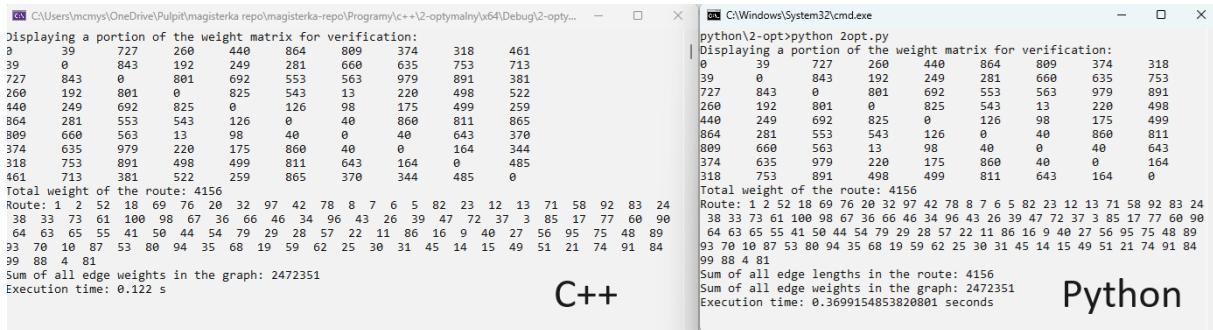
disp('Sum of all edge weights in the graph:');
disp(TotalWeight);

elapsedTime = toc;
fprintf('Execution time: %.3f s\n', elapsedTime);
end

```

Rysunek 76. Funkcja main() zaimplementowana w Matlabie.
[źródło: opracowanie własne]

Weryfikacja poprawności opracowanych algorytmów 2-opt w językach C++, Python oraz Matlab została przeprowadzona poprzez badanie na danych wejściowych wygenerowanych przy użyciu generatora liczb losowych. Rezultaty działania algorytmów zaimplementowanych w języku C++ oraz Python zaprezentowano na rysunku 77. Z kolei efekt działania programu w Matlabie przedstawiony został na rysunku 78.



```

C:\Users\mcmys\OneDrive\Pulpit\magisterka repo\magisterka-repo\Programy\c++\2-opt\mainer\x64\Debug\2-opt...
python2-opt>python 2opt.py
python2-opt>python 2opt.py
| python2-opt>python 2opt.py
| Displaying a portion of the weight matrix for verification:
| 0 39 727 260 440 864 809 374 318 461
| 39 0 843 192 249 281 660 635 753 713
| 727 843 0 801 692 553 563 979 891 381
| 260 192 801 0 825 543 13 220 498 522
| 440 249 692 825 0 126 98 175 499 259
| 864 281 553 543 126 0 40 860 811 865
| 809 660 563 13 98 40 0 40 643 370
| 374 635 979 220 175 860 40 0 164 344
| 318 753 891 498 499 811 643 164 0 485
| 461 713 381 522 259 865 370 344 485 0
| Total weight of the route: 4156
| Route: 1 2 52 18 69 76 20 32 97 42 78 8 7 6 5 82 23 12 13 71 58 92 83 24
| 38 33 73 61 100 98 67 36 66 46 34 96 43 26 39 47 72 37 3 85 17 77 60 90
| 64 63 65 55 41 50 44 54 79 29 28 57 22 11 86 16 9 40 27 56 95 75 48 89
| 93 70 10 87 53 80 94 35 68 19 59 62 25 30 31 45 14 15 49 51 21 74 91 84
| 99 88 4 81
| Sum of all edge weights in the graph: 2472351
| Execution time: 0.122 s
C++                                         Python

```

Rysunek 77. Przedstawienie wyników działania programów C++ oraz Python.
[źródło: opracowanie własne]

```

Command Window
>> dwuopt
Displaying a portion of the weight matrix for verification:
0 39 727 260 440 864 809 374 318 461
39 0 843 192 249 281 660 635 753 713
727 843 0 192 249 281 660 635 753 713
260 192 192 0 249 318 461 440 864 809
440 249 692 825 0 126 98 175 499 259
864 281 553 543 126 0 40 860 811 865
809 660 563 13 98 40 0 40 643 370
374 635 979 220 175 864 40 0 164 344
318 753 891 494 499 811 643 164 0 485
461 713 381 522 259 865 370 344 485 0
Total weight of the route:
4156

Route:
Columns 1 through 34
1 2 52 18 69 76 20 32 97 42 78 8 7 6 5 82 23 12 13 71 58 92 83 24 38 33 73 61 100 98 67 36 66 46
Columns 35 through 68
34 96 43 26 39 47 72 37 3 85 17 77 60 90 64 63 65 55 41 50 44 54 79 29 28 57 22 11 86 16 9 40 27 56
Columns 69 through 100
95 75 48 89 93 70 10 87 53 80 94 35 68 19 59 62 25 30 31 45 14 15 49 51 21 74 91 84 99 88 4 81

Sum of all edge lengths in the route:
4156

Sum of all edge weights in the graph:
2472351

Execution time: 0.047 s
f8 >>

```

Rysunek 78. Przedstawienie wyniku działania programu opracowanego w Matlabie.
[źródło: opracowanie własne]

Wyniki wszystkich trzech programów przedstawione na rysunkach 77 oraz 78 okazały się zgodne pod względem przeprowadzonych obliczeń. Całkowita waga trasy wyniosła 4156, natomiast suma wszystkich wag krawędzi w grafie wyniosła 2472351. Różnice pojawiły się w czasach wykonania programów. W wyniku przeprowadzonego badania najszybsza okazała się implementacja wykonana w Matlabie, uzyskując wynik 0.047 sekundy. Na pozycji drugiej znalazł się kod napisany w języku C++, którego wynik czasowy wyniósł 0.122 sekundy. Ostatni i zarazem najwolniejszy okazał się kod zaimplementowany w języku Python, którego czas działania wyniósł 0.370 sekundy. Przedstawione wyniki potwierdzają poprawność działania algorytmu 2-opt w każdym z języków.

6.4. Implementacja algorytmu trój-optymalnego

W ramach implementacji wybranych algorytmów komputerowych zadecydowano o konieczności opracowania również algorytmu trój-optymalnego. W części badawczej pracy dyplomowej dokonano implementacji tego algorytmu zarówno w językach C++, Python oraz Matlab, aby porównać uzyskane wyniki.

Implementacja rozpoczęta została od deklaracji zmiennych globalnych, które stanowiły niezbędny element do prawidłowego działania algorytmu. Zmienna N przechowywała liczbę wierzchołków, z kolei M odpowiedzialna była za reprezentacje ilości krawędzi. Wskaźnik Ptr wykorzystany został do stworzenia tablicy śledzącej połączenia pomiędzy wierzchołkami. Dwuwymiarowa tablica W zaimplementowana została w celu przechowywania wag krawędzi pomiędzy wierzchołkami grafu. Z kolei zmienna Route wykorzystana została

do przechowywania aktualnej trasy, którą wyliczał program. Podczas implementacji zdefiniowane zostały również zmienne pomocnicze tj. Ahead, I, J, K, Index, Limit i Max, które wykorzystane zostały podczas różnych etapów obliczeniowych. Tweight oraz TotalWeight stanowiły zmienne przechowujące odpowiednio całkowitą wagę trasy, jak i sumę wag wszystkich krawędzi w grafie. Aby ułatwić operację zamian krawędzi, zastosowana została enumeracja SwapType, która definiowała dwa typy zamiany: asymetryczny Asym oraz symetryczny Sym. Struktura SwapData z kolei, zaimplementowana został w celu przechowywania danych dotyczących aktualnie rozważanej zmiany, w tym również wierzchołki biorące udział w zamianie tj. X1, X2, Y1, Y2, Z1, Z2 oraz potencjalny zysk z przeprowadzonej zamiany Gain. Dodatkowo SwapData przechowuje również typ zamiany Choice. Zmienne strukturalne BestSwap oraz Swap wykorzystane zostały do przechowywania danych najlepszej zamiany oraz aktualnie rozważanej zmiany. Fragment kodu odpowiedzialny za proces deklaracji zmiennych globalnych przedstawiony został na rysunku 79.

```

int N, M;
int* Ptr;
int** W;
int* Route;
int Ahead, I, J, K, Index, Limit, Max;
int Tweight = 0;
int TotalWeight = 0;
enum SwapType { Asym, Sym };
struct SwapData
{
    int X1, X2, Y1, Y2, Z1, Z2, Gain;
    SwapType Choice;
} BestSwap, Swap;

```

Rysunek 79. Inicjalizacja zmiennych globalnych w języku C++.
[źródło: opracowanie własne]

W odróżnieniu od języka C++, w Pythonie zmienne globalne zdefiniowane i zainicjalizowane zostały bezpośrednio w funkcjach, w których zostały używane. Język Python nie wymaga deklarowania typu zmiennych, ze względu na fakt, że jest językiem dynamicznie typowanym. Fragment kodu zawierający inicjalizację klas SwapType oraz SwapData został zaprezentowany na rysunku 80.

```

class SwapType:
    Asym, Sym = range(2)

class SwapData:
    def __init__(self):
        self.X1 = self.X2 = self.Y1 = self.Y2 = self.Z1 = self.Z2 = self.Gain = 0
        self.Choice = SwapType.Asym

```

Rysunek 80. Inicjalizacja klas SwapType oraz SwapData w języku Python
[źródło: opracowanie własne]

W przypadku kodu napisanego w Matlabie również nie występowała potrzeba inicjalizacji zmiennych globalnych na początku pliku. Podobnie jak w języku Python, zmienne zadeklarowane zostały wewnątrz funkcji lub skryptów. Z tego powodu postanowiono przejść do kolejnego etapu implementacji.

W kolejnym kroku implementacji algorytmu utworzona została funkcja SwapCheck. Została ona zaimplementowana w celu oceny oraz aktualizacji danych dotyczących zamiany wierzchołków w grafie. Rozpoczęto on inicjalizacji zmiennej Gain w strukturze Swap o wartości wynoszącej 0. W kolejnym kroku obliczono sumaryczną wagę krawędzi pomiędzy wierzchołkami X1, X2, Y1, Y2, Z1, Z2. Uzyskany wynik obliczeń przypisano do zmiennej DelWeight. Następnie dokonano sprawdzenia dwóch możliwych typów zamiany tj. asymetrycznego oraz symetrycznego. W przypadku zamiany asymetrycznej obliczony został maksymalny zysk, który porównany został do aktualnego. Jeżeli nowa wartość była większa, dokonano aktualizacji zmiennej Gain oraz ustalonio typ zamiany na asymetryczny. W analogiczny sposób dokonano obliczenia dla zamiany symetrycznej. Rozpoczęto od wyliczenia nowego zysku, następnie porównano go z aktualnym, dokonując aktualizacji w razie potrzeby oraz ustawiając typ zmiany na symetryczny. Dzięki temu rozwiązaniu funkcja oceniła i wybrała najbardziej korzystną zamianę wierzchołków. Fragment kodu w języku C++ odpowiedzialny za ten proces przedstawiony został na rysunku 81.

```

void SwapCheck(SwapData& Swap)
{
    int DelWeight, Max;
    Swap.Gain = 0;
    DelWeight = W[Swap.X1][Swap.X2] + W[Swap.Y1][Swap.Y2] + W[Swap.Z1][Swap.Z2];
    Max = DelWeight - (W[Swap.Y1][Swap.X1] + W[Swap.Z1][Swap.X2] + W[Swap.Z2][Swap.Y2]);
    if (Max > Swap.Gain) Swap.Gain = Max, Swap.Choice = Asym;
    Max = DelWeight - (W[Swap.X1][Swap.Y2] + W[Swap.Z1][Swap.X2] + W[Swap.Y1][Swap.Z2]);
    if (Max > Swap.Gain) Swap.Gain = Max, Swap.Choice = Sym;
}

```

Rysunek 81. Funkcja SwapCheck w języku C++
[źródło: opracowanie własne]

W sposób analogiczny funkcja SwapCheck() została zaimplementowana w języku Python. Cechą odróżniającą tę implementację od wersji w języku C++ było zdefiniowanie typu wyliczeniowego SwapType jako klasy, której atrybutami są wartości wyliczeniowe. Użycie tych wartości wymagało odniesienia do klasy, np. SwapType.Asym. Fragment kodu w języku Python odpowiedzialny za klasę SwapCheck() został zaprezentowany na rysunku 82.

```
def SwapCheck(Swap, W):
    DelWeight = W[Swap.X1][Swap.X2] + W[Swap.Y1][Swap.Y2] + W[Swap.Z1][Swap.Z2]
    Max = DelWeight - (W[Swap.Y1][Swap.X1] + W[Swap.Z1][Swap.X2] + W[Swap.Z2][Swap.Y2])
    if Max > Swap.Gain:
        Swap.Gain = Max
        Swap.Choice = SwapType.Asym
    Max = DelWeight - (W[Swap.X1][Swap.Y2] + W[Swap.Z1][Swap.X2] + W[Swap.Y1][Swap.Z2])
    if Max > Swap.Gain:
        Swap.Gain = Max
        Swap.Choice = SwapType.Sym
```

Rysunek 82. Klasa SwapCheck() w języku Python
[źródło: opracowanie własne]

W podobny sposób zaimplementowana została funkcja SwapCheck() w Matlabie. Implementacja ta różni się od swoich odpowiedników w C++ i Pythonie pod względem składni oraz typowania. W Matlabie typy wyliczeniowe nie zostały wykorzystane, a zamiast tego zastosowano ciągów znaków do reprezentacji wyborów zamiany. W kontekście składni oraz typowania Matlab podobny był do Pythona pod względem dynamicznego typowania, jednak różnicą była własna składnia do operacji na macierzach. Fragment kodu odpowiedzialny za omówioną funkcję został przedstawiony na rysunku 83.

```
function Swap = SwapCheck(Swap, W)
    DelWeight = W(Swap.X1, Swap.X2) + W(Swap.Y1, Swap.Y2) + W(Swap.Z1, Swap.Z2);

    MaxGain = DelWeight - (W(Swap.Y1, Swap.X1) + W(Swap.Z1, Swap.X2) + W(Swap.Z2, Swap.Y2));
    if MaxGain > Swap.Gain
        Swap.Gain = MaxGain;
        Swap.Choice = "Asym";
    end

    MaxGain = DelWeight - (W(Swap.X1, Swap.Y2) + W(Swap.Z1, Swap.X2) + W(Swap.Y1, Swap.Z2));
    if MaxGain > Swap.Gain
        Swap.Gain = MaxGain;
        Swap.Choice = "Sym";
    end
end
```

Rysunek 83. Funkcja SwapCheck() przedstawiona w Matlabie
[źródło: opracowanie własne]

W dalszej części przeprowadzania implementacji algorytmu trójoptymalnego zaimplementowana została funkcja Reverse. Opracowana ona została w celu odwrócenia kolejności elementów w liście połączonej, rozpoczynając od węzła start, a kończąc na finish. Rozpoczęto od sprawdzenia czy węzły początkowy oraz końcowy są różne. W przypadku, gdy

okazały się inne, zainicjalizowane zostały zmienne last oraz next odpowiednio wykorzystując wartość start oraz wskaźnik na następny węzeł. W kolejnym etapie, w pętli do-while, wykonane zostało odwracanie wskaźników dla każdego węzła między start a finish, aktualizując wskaźnik next do kolejnego węzła oraz przypisując wskaźnik poprzedniego węzła do bieżącego. Proces kontynuowano, dopóki last nie osiągnęło węzła finish. Fragment kodu odpowiedzialny za tą funkcję przedstawiony został na rysunku 84.

```
void Reverse(int start, int finish)
{
    int ahead, last, next;
    if (start != finish) last = start, next = Ptr[last];
    do ahead = Ptr[next], Ptr[next] = last, last = next, next = ahead; while (last != finish);
}
```

Rysunek 84. Funkcja Reverse() w języku C++
[źródło: opracowanie własne]

W analogiczny sposób funkcja Reverse() została zaimplementowana w języku Python, co zostało przedstawione na rysunku 85.

```
def Reverse(Ptr, start, finish):
    if start != finish:
        last = start
        next = Ptr[last]
        while last != finish:
            ahead = Ptr[next]
            Ptr[next] = last
            last = next
            next = ahead
```

Rysunek 85. Funkcja Reverse() w języku Python.
[źródło: opracowanie własne]

Funkcja Reverse() zaimplementowana została również w programie napisanym w Matlabie. Używa składni podobnej do Pythona, ale z własnymi operatorami i strukturą. Używa operatora $\sim=$ dla porównań oraz nawiasów okrągłych do indeksowania tablic. Fragment kodu omawianej funkcji przedstawiony został na rysunku 86.

```

function Ptr = Reverse(Ptr, start, finish)
    if start ~= finish
        last = start;
        next = Ptr(last);
        while last ~= finish
            ahead = Ptr(next);
            Ptr(next) = last;
            last = next;
            next = ahead;
        end
    end
end

```

Rysunek 86. Funkcja Reverse() opracowana w Matlabie.

[źródło: opracowanie własne]

W kolejnym etapie zaimplementowana została funkcja Initialization. Jej celem było wczytanie danych z pliku, zainicjalizowanie macierzy wag oraz tablic pomocniczych. Najpierw otwarto plik wejściowy i wczytano liczby N i M, które oznaczają odpowiednio liczbę wierzchołków i krawędzi. Następnie zaalokowano pamięć dla tablic Ptr, Route oraz macierzy W. Dla każdej krawędzi wczytano wierzchołki X i Y oraz wagę Temp, po czym zaktualizowano macierz wag W oraz sumaryczną wagę wszystkich krawędzi TotalWeight. Ustaloną początkową trasę w tablicy Route, ustawiając wagi pętli wierzchołków na zero. Na koniec obliczono wstępную sumaryczną wagę trasy Tweight, sumując wagi kolejnych odcinków trasy i zamykając plik. Fragment kodu C++ odpowiedzialny za funkcję Initialization przedstawiony został na rysunku 87.

```

void Initialization()
{
    int X, Y, Temp;
    ifstream data("C:\\\\Users\\\\mcmys\\\\OneDrive\\\\Pulpit\\\\magisterka_repo\\\\magisterka-repo\\\\Programy\\\\testy\\\\edges100.in");
    data >> N >> M;
    Ptr = new int[N + 1];
    Route = new int[N + 1];
    W = new int* [N + 1];
    for (int i = 0; i <= N; ++i) W[i] = new int[N + 1];
    for (int i = 1; i <= M; ++i)
    {
        data >> X >> Y >> Temp;
        W[X][Y] = Temp;
        W[Y][X] = Temp;
        TotalWeight += Temp;
    }
    for (int i = 1; i <= N; ++i) Route[i] = i, W[i][i] = 0;
    for (int i = 1; i < N; ++i) Tweight += W[i][i + 1];
    Tweight += W[N][1];
    data.close();
}

```

Rysunek 87. Funkcja Initialization() w języku C++.

[źródło: opracowanie własne]

Funkcja Initialization w Pythonie działa podobnie do swojej implementacji w języku C++, ale różni się drobnymi szczegółami. W Pythonie plik otwarty został za pomocą kontekstu

menedżera with, co zapewnia automatyczne zamknięcie pliku po zakończeniu operacji. Dane zostały wczytywane linia po linii metodą readline() i przetworzone funkcją map(). Kolejna różnica dotyczy inicjalizacji macierzy „W”, która w Pythonie została utworzona funkcją np.zeros() z biblioteki numpy, wypełniając ją zerami. Fragment kodu odpowiedzialny za omówioną funkcję przedstawiony został na rysunku 88.

```
def Initialization():
    global N, M, Ptr, W, Route, Tweight, TotalWeight

    with open('C:\\Users\\mcmys\\OneDrive\\Pulpit\\magisterka_repo\\magisterka-repo\\Programy\\testy\\edges100.in') as data:
        N, M = map(int, data.readline().split())
        Ptr = [0] * (N + 1)
        Route = [0] * (N + 1)
        W = np.zeros((N + 1, N + 1), dtype=int)
        Tweight = 0
        TotalWeight = 0

        for _ in range(M):
            X, Y, Temp = map(int, data.readline().split())
            W[X][Y] = Temp
            W[Y][X] = Temp
            TotalWeight += Temp

        for i in range(1, N + 1):
            Route[i] = i
            W[i][i] = 0

        for i in range(1, N):
            Tweight += W[i][i + 1]
        Tweight += W[N][1]
```

Rysunek 88. Funkcja Initialization() w języku Python.
[źródło: opracowanie własne]

Funkcja Initialization () zaimplementowana została również w programie napisanym w Matlabie. Program ten w odróżnieniu od C++ oraz Pythona wykorzystuje fopen do otwierania plików i fclose do ich zamykania. Operacje odczytu są wykonywane za pomocą fscanf. Fragment kodu odpowiedzialny za omówioną funkcję został przedstawiony na rysunku 89.

```
function [N, M, W, Route, TotalWeight] = Initialization()
    fid = fopen('C:\\Users\\mcmys\\OneDrive\\Pulpit\\magisterka_repo\\magisterka-repo\\Programy\\testy\\edges100.in', 'r');
    if fid == -1
        error('File not found');
    end

    data = fscanf(fid, '%d %d', 2);
    N = data(1);
    M = data(2);

    W = zeros(N, N);
    Route = 1:N;
    TotalWeight = 0;

    for i = 1:M
        edge = fscanf(fid, '%d %d %d', 3);
        X = edge(1);
        Y = edge(2);
        Temp = edge(3);
        W(X, Y) = Temp;
        W(Y, X) = Temp;
        TotalWeight = TotalWeight + Temp; % Add edge weight to total sum
    end

    fclose(fid);
end
```

Rysunek 89. Funkcja Initialization() w języku Matlabie.
[źródło: opracowanie własne]

W kolejnym kroku dokonano implementacji funkcji Calculations(), której celem była optymalizacja trasy w grafie. Na początku ustawiono wskaźniki w tablicy Ptr tak, aby wskazywały na kolejne wierzchołki trasy. Następnie w pętli do-while zainicjalizowano zmienną BestSwap.Gain na 0 i rozpoczęto proces przeglądania możliwych zamian wierzchołków w grafie. W zagnieżdżonych pętlach for sprawdzano wszystkie możliwe zamiany trójkę wierzchołków, korzystając z funkcji SwapCheck, aby obliczyć zysk z zamiany. Jeśli znaleziono lepszą zamianę, aktualizowano zmienne BestSwap. W przypadku odnalezienia najlepszej zamiany, odwracano odpowiednie wskaźniki w tablicy Ptr i aktualizowano wagę trasy Tweight. Proces ten kontynuowano do momentu, aż nie znaleziono już korzystniejszych zamian. Na końcu odtworzono trasę w tablicy Route na podstawie wskaźników w Ptr. Fragment kodu odpowiedzialny za omówioną funkcję przedstawiony został na rysunku 90.

```

void Calculations()
{
    for (I = 1; I < N; ++I) Ptr[Route[I]] = Route[I + 1];
    Ptr[Route[N]] = Route[1];
    do
    {
        BestSwap.Gain = 0, Swap.X1 = 1;
        for (int I = 1; I <= N; ++I)
        {
            Swap.X2 = Ptr[Swap.X1], Swap.Y1 = Swap.X2;
            for (J = 2; J <= N - 3; ++J)
            {
                Swap.Y2 = Ptr[Swap.Y1], Swap.Z1 = Ptr[Swap.Y2];
                for (K = J + 2; K <= N - 1; ++K)
                {
                    Swap.Z2 = Ptr[Swap.Z1];
                    SwapCheck(Swap);
                    if (Swap.Gain > BestSwap.Gain) BestSwap = Swap;
                    Swap.Z1 = Swap.Z2;
                }
                Swap.Y1 = Swap.Y2;
            }
            Swap.X1 = Swap.X2;
        }
        if (BestSwap.Gain > 0)
        {
            if (BestSwap.Choice == Asym)
            {
                Reverse(BestSwap.Z2, BestSwap.X1);
                Ptr[BestSwap.Y1] = BestSwap.X1, Ptr[BestSwap.Z2] = BestSwap.Y2;
            }
            else Ptr[BestSwap.X1] = BestSwap.Y2, Ptr[BestSwap.Y1] = BestSwap.Z2;
            Ptr[BestSwap.Z1] = BestSwap.X2;
            Tweight -= BestSwap.Gain;
        }
    } while (BestSwap.Gain);
    Index = 1;
    for (int I = 1; I <= N; ++I) Route[I] = Index, Index = Ptr[Index];
}

```

Rysunek 90. Funkcja Calculations() w języku C++.
[źródło: opracowanie własne]

W sposób analogiczny funkcja Calculations zaimplementowana została w języku Python, co zostało przedstawione na rysunku 91.

```

def Calculations():
    global Ptr, Route, Tweight

    for I in range(1, N):
        Ptr[Route[I]] = Route[I + 1]
    Ptr[Route[N]] = Route[1]

    while True:
        BestSwap = SwapData()
        Swap = SwapData()
        Swap.X1 = 1

        for I in range(1, N + 1):
            Swap.X2 = Ptr[Swap.X1]
            Swap.Y1 = Swap.X2

            for J in range(2, N - 1):
                Swap.Y2 = Ptr[Swap.Y1]
                Swap.Z1 = Ptr[Swap.Y2]

                for K in range(J + 2, N + 1):
                    Swap.Z2 = Ptr[Swap.Z1]
                    SwapCheck(Swap, W)

                    if Swap.Gain > BestSwap.Gain:
                        BestSwap = SwapData()
                        BestSwap.X1, BestSwap.X2 = Swap.X1, Swap.X2
                        BestSwap.Y1, BestSwap.Y2 = Swap.Y1, Swap.Y2
                        BestSwap.Z1, BestSwap.Z2 = Swap.Z1, Swap.Z2
                        BestSwap.Gain, BestSwap.Choice = Swap.Gain, Swap.Choice
                        Swap.Z1 = Swap.Z2

                    Swap.Y1 = Swap.Y2

                    Swap.X1 = Swap.X2

                    if BestSwap.Gain > 0:
                        if BestSwap.Choice == SwapType.Asym:
                            Reverse(Ptr, BestSwap.Z2, BestSwap.X1)
                            Ptr[BestSwap.Y1] = BestSwap.X1
                            Ptr[BestSwap.Z2] = BestSwap.Y2
                        else:
                            Ptr[BestSwap.X1] = BestSwap.Y2
                            Ptr[BestSwap.Y1] = BestSwap.Z2
                            Ptr[BestSwap.Z1] = BestSwap.X2
                            Tweight -= BestSwap.Gain
                    else:
                        break

        Index = 1
        for I in range(1, N + 1):
            Route[I] = Index
            Index = Ptr[Index]

```

Rysunek 91. Funkcja Calculations() w języku Python.

[źródło: opracowanie własne]

Funkcja Calculations() została również zaimplementowana w Matlabie. Cechą odróżniającą tą implementację od innych to wykorzystanie fopen i fscanf do odczytu danych z plików. Fragment kodu odpowiedzialny implementacji funkcji przedstawiony został na rysunku 92.

```

function [Tweight, Route] = Calculations(N, W, Route)
    Ptr = zeros(1, N);
    for i = 1:N-1
        Ptr(Route(i)) = Route(i+1);
    end
    Ptr(Route(N)) = Route(1);

    Tweight = sum(W(sub2ind(size(W), Route(1:end-1), Route(2:end)))); % Sum of weights for all edges except the last one
    Tweight = Tweight + W(Route(N), Route(1));

    BestSwap.Gain = 0;
    BestSwap.X1 = 0; BestSwap.X2 = 0;
    BestSwap.Y1 = 0; BestSwap.Y2 = 0;
    BestSwap.Z1 = 0; BestSwap.Z2 = 0;
    BestSwap.Choice = '';

    while true
        BestSwap.Gain = 0;
        Swap.X1 = 1;
        for i = 1:N
            Swap.X2 = Ptr(Swap.X1);
            Swap.Y1 = Swap.X2;
            for j = 2:N-3
                Swap.Y2 = Ptr(Swap.Y1);
                Swap.Z1 = Ptr(Swap.Y2);
                for k = j+2:N-1
                    Swap.Z2 = Ptr(Swap.Z1);
                    Swap.Gain = 0;
                    Swap.Choice = '';
                    Swap = SwapCheck(Swap, W);
                    if Swap.Gain > BestSwap.Gain
                        BestSwap = Swap;
                    end
                    Swap.Z1 = Swap.Z2;
                end
                Swap.Y1 = Swap.Y2;
            end
            Swap.X1 = Swap.X2;
        end

        if BestSwap.Gain <= 0
            break;
        end

        if BestSwap.Choice == "Asym"
            Ptr = Reverse(Ptr, BestSwap.Z2, BestSwap.X1);
            Ptr(BestSwap.Y1) = BestSwap.X1;
            Ptr(BestSwap.Z2) = BestSwap.Y2;
        else
            Ptr(BestSwap.X1) = BestSwap.Y2;
            Ptr(BestSwap.Y1) = BestSwap.Z2;
        end
        BestSwap.Z1 = BestSwap.X2;
        Tweight = Tweight - BestSwap.Gain;
    end

    Index = 1;
    for i = 1:N
        Route(i) = Index;
        Index = Ptr(Index);
    end
end

```

Rysunek 92. Implementacja funkcji Calculations w Matlabie.
[źródło: opracowanie własne]

W kolejnym etapie implementacji dokonano inicjalizacji funkcji SaveResultsToCSV. Zadaniem tej funkcji jest zapis uzyskanych wyników do pliku CSV. Otworzono strumień wyjściowy do pliku o nazwie przekazanej jako argument filename. Następnie zapisano

nagłówki kolumn "Node" i "NextNode". W pętli for zapisano kolejne węzły trasy oraz ich następcy, pobierając odpowiednie wartości z tablic Route i Ptr. Po zakończeniu pętli zapisano całkowitą wagę trasy Tweight, sumę wszystkich wag krawędzi w grafie TotalWeight oraz czas wykonania programu duration w sekundach. Na końcu zamknięto strumień wyjściowy, kończąc operację zapisu. Implementacja funkcji zapisującej dane do pliku CSV został przedstawiony na rysunku 93.

```
void SaveResultsToCSV(const string& filename, double duration)
{
    ofstream csvFile(filename);
    csvFile << "Node,NextNode\n";
    for (int i = 1; i <= N; ++i)
    {
        csvFile << Route[i] << "," << Ptr[Route[i]] << "\n";
    }
    csvFile << "\nTotal weight of the route," << Tweight << "\n";
    csvFile << "Sum of all edge weights in the graph," << TotalWeight << "\n";
    csvFile << "Execution time (seconds)," << duration << "\n";
    csvFile.close();
}
```

Rysunek 93. Implementacja funkcji SaveResultsToCSV w języku C++.
[źródło: opracowanie własne]

W sposób analogiczny dokonano implementacji zapisu danych do pliku CSV w języku Python, co zostało przedstawione na rysunku 94.

```
def SaveResultsToCSV(filename, N, Route, Tweight, TotalWeight):
    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['Node', 'NextNode'])
        for i in range(1, N):
            writer.writerow([Route[i], Route[i + 1]])
        writer.writerow([Route[N], Route[1]]) # Last edge
        writer.writerow([])
        writer.writerow(['Total weight of the route', Tweight])
        writer.writerow(['Sum of all edge weights in the graph', TotalWeight])
```

Rysunek 94. Implementacja funkcji SaveResultsToCSV w języku Python.
[źródło: opracowanie własne]

Funkcja SaveResultsToCSV została zaimplementowana również w Matlabie. Zostało to przedstawione na rysunku 95.

```

function SaveResultsToCSV(filename, N, Route, Tweight, TotalWeight)
    fid = fopen(filename, 'w');
    if fid == -1
        error('Could not create CSV file');
    end

    fprintf(fid, 'Node,NextNode\n');
    for i = 1:N
        if i < N
            fprintf(fid, '%d,%d\n', Route(i), Route(i+1));
        else
            fprintf(fid, '%d,%d\n', Route(i), Route(1));
        end
    end
    fprintf(fid, '\nTotal weight of the route,%d\n', Tweight);
    fprintf(fid, 'Sum of all edge weights in the graph,%d\n', TotalWeight);

    fclose(fid);
end

```

Rysunek 95. Implementacja funkcji SaveResultsToCSV w Matlabie.

[źródło: opracowanie własne]

W ostatniej fazie implementacji algorytmu trójoptymalnego przedstawiona została również funkcję główną programu. Na początku zmierzono czas rozpoczęcia wykonywania programu przy użyciu funkcji `high_resolution_clock::now()`. Następnie wywołano funkcje `Initialization` i `Calculations`, które odpowiedzialne są za inicjalizację danych oraz optymalizację trasy w grafie. Po wykonaniu obliczeń wyświetlono całkowitą wagę trasy `Tweight` oraz szczegóły trasy poprzez iterację i wyświetlenie każdego węzła z tablicy `Route`. Następnie wyświetlono sumę wszystkich wag krawędzi w grafie `TotalWeight`. Zmierzono czas zakończenia wykonywania programu i obliczono czas trwania operacji w sekundach. Wynik ten został wyświetlony na ekranie. Na końcu wyniki zostały zapisane do pliku CSV przy użyciu funkcji `SaveResultsToCSV`, a program zakończył swoje działanie po oczekiwaniu na naciśnięcie klawisza przez użytkownika. Funkcja główna programu przedstawiona została na rysunku 96.

```

int main()
{
    auto start = high_resolution_clock::now();

    Initialization();
    Calculations();
    cout << "Total weight of the route: " << Tweight << endl;
    cout << "Route: ";
    for (int i = 1; i <= N; ++i) cout << Route[i] << " ";
    cout << endl;

    cout << "Sum of all edge weights in the graph: " << TotalWeight << endl;

    auto end = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end - start).count();
    double duration_seconds = duration / 1000.0;
    cout << "Execution time: " << duration_seconds << " seconds" << endl;

    SaveResultsToCSV("results.csv", duration_seconds);

    cin.get();
    return 0;
}

```

Rysunek 96. Implementacja funkcji głównej w języku C++.

[źródło: opracowanie własne]

W analogiczny sposób funkcja główna programu zaimplementowana z wykorzystaniem języka Python. Przedstawione to zostało na rysunku 97.

```
if __name__ == "__main__":
    start = time.time()

    Initialization()
    Calculations()
    print(f"Total weight of the route: {Tweight}")
    print("Route: ", end="")
    for i in range(1, N + 1):
        print(Route[i], end=" ")
    print()

    print(f"Sum of all edge weights in the graph: {TotalWeight}")

    end = time.time()
    duration = end - start
    print(f"Execution time: {duration:.3f} seconds")

    SaveResultsToCSV('results.csv', N, Route, Tweight, TotalWeight)
```

Rysunek 97. Implementacja funkcji głównej w języku Python.
[źródło: opracowanie własne]

Funkcja główna programu zaimplementowana została w programie Matlab, co zostało przedstawione na rysunku 98.

```
function main
    startTime = tic;

    [N, M, W, Route, TotalWeight] = Initialization();

    [Tweight, Route] = Calculations(N, W, Route);

    fprintf('Total weight of the route: %d\n', Tweight);
    fprintf('Route: ');
    fprintf('%d ', Route);
    fprintf('\n');

    fprintf('Sum of all edge weights in the graph: %d\n', TotalWeight);

    duration = toc(startTime);
    fprintf('Execution time: %.3f seconds\n', duration);

    SaveResultsToCSV('results.csv', N, Route, Tweight, TotalWeight);
end
```

Rysunek 98. Implementacja funkcji głównej w Matlabie.
[źródło: opracowanie własne]

W celu potwierdzenia poprawności działania opracowanych algorytmów trójoptymalnych w języku C++, Python oraz Matlab przeprowadzono badanie na wygenerowanych podczas przedstawiania działania generatora liczb losowych danych wejściowych. Efekt działania przedstawiony został na rysunku 99.

```

C++ Terminal:
Total weight of the route: 2126
Route: 1 96 34 100 2 45 89 86 99 88 58 44 74 91 80 84 6 41 55 58 63
65 94 35 7 4 77 17 19 68 42 97 95 56 22 57 15 14 61 73 26 43 87 10 9
75 48 23 28 29 27 66 46 85 3 37 52 18 49 38 33 21 51 71 93 78 40 9
69 90 8 78 81 72 59 11 30 31 92 83 24 13 98 67 32 25 62 16 64 39
Sum of all edge weights in the graph: 2472351
Execution time: 0.868 seconds

Python Terminal:
Total weight of the route: 2126
Route: 1 96 34 100 2 45 89 86 99 88 58 44 74 91 80 84 6 41 55 58 63
58 63 65 94 35 7 4 77 17 19 68 42 97 95 56 22 57 15 14 61 73 26 43 87 10 84
58 63 65 94 35 7 4 77 17 19 68 42 97 95 56 22 57 15 14 61 80 84 6 41 55
24 13 98 67 32 25 62 16 64 39 47 5 37 52 18 49 38
33 21 51 71 93 80 8 78 81 72 59 11 30 31 92 83
24 13 98 67 32 25 62 16 64 39 47 5 37 52 18 49 38
28
Sum of all edge weights in the graph: 2472351
Execution time: 73.082 seconds

Matlab Command Window:
>> trzyopt
Total weight of the route: 2126
Route: 1 96 34 100 2 45 89 86 99 88 58 44 74 91 80 84 6 41 55 58 63 65 94 35 7 4 77 17 19 68 42 97 95 56 22 57 15 14 61 73 26 43 87 10 75 48 23 28 29 27 66 46 85 3 37 52 18 49 38
Execution time: 52.837 seconds
f<>

```

Rysunek 99. Efekt działania programu w C++, Pythonie oraz Matlabie.
[źródło: opracowanie własne]

Rysunek 99 przedstawia wyniki uzyskane w wyniku przeprowadzenia badania na danych wejściowych dla 100 wierzchołków. Programy zgodnie uzyskały wynik optymalizacji trasy równy 2126, a także taką samą sumę wag krawędzi, która wyniosła 2472351. Jedyną różnicą pomiędzy programami jest czas działania programu. Najkrótszy czas uzyskany został w algorytmie zaimplementowanym w języku C++ i wyniósł on 0.868 sekundy. Na pozycji drugiej znajduje się wynik uzyskany przez kod zaimplementowany w Matlabie (52.837 sekundy). Ostatnie miejsce z czasem 73.082 sekundy uzyskał program opracowany w języku Python.

6.5. Implementacja algorytmu Christofidesa

W ramach implementacji wybranych algorytmów komputerowych zadecydowano o konieczności opracowania również algorytmu Christofidesa. W części badawczej pracy dyplomowej dokonano implementacji tego algorytmu zarówno w językach C++, Python oraz Matlab, aby porównać uzyskane wyniki.

Algorytm Christofidesa jest heurystycznym podejściem do rozwiązywania problemu komiwojażera. W zaimplementowanych kodzie program wykorzystywał różne struktury danych oraz funkcje pomocnicze, w celu znalezienia trasy komiwojażera o możliwie minimalnym koszcie.

Implementacje rozpoczęto od zdefiniowania struktury „Graf” reprezentującej graf z wykorzystaniem macierzy sąsiedztwa „adjMatrix” przechowującej wagi krawędzi pomiędzy wierzchołkami. Konstruktor „Graf” dokonał inicjalizacji liczby wierzchołków oraz dokonuje utworzenia macierzy sąsiedztwa o wymiarach $V \times V$, która wypełniona została zerami. Funkcja „addEdge” wykorzystana została do dodania krawędzi pomiędzy wierzchołkami „u” oraz „v” o wadze wynoszącej „w”, aktualizując przy tym odpowiednie wartości w macierzy sąsiedztwa

w obu kierunkach, co umożliwiło reprezentację grafu nieskierowanego¹⁴. Fragment kodu odnoszący się do omówionej implementacji został przedstawiony na rysunku 100.

```
struct Graph {  
    int V;  
    std::vector<std::vector<double>> adjMatrix;  
  
    Graph(int V) : V(V), adjMatrix(V, std::vector<double>(V, 0)) {}  
  
    void addEdge(int u, int v, double w) {  
        adjMatrix[u][v] = w;  
        adjMatrix[v][u] = w;  
    }  
};
```

Rysunek 100. Implementacja struktury Graph w języku C++.
[źródło: opracowanie własne]

W sposób analogiczny zaimplementowana została klasa Graph za pomocą języka Python. Cechą odróżniającą tego kodu jest wykorzystanie w konstruktorze metody „__init__”. Fragment kodu implementacji klasy Graph w Pythonie przedstawiony został na rysunku 101.

```
class Graph:  
    def __init__(self, V):  
        self.V = V  
        self.adjMatrix = [[0] * V for _ in range(V)]  
  
    def add_edge(self, u, v, w):  
        self.adjMatrix[u][v] = w  
        self.adjMatrix[v][u] = w
```

Rysunek 101. Implementacja klasy Graph w języku Python
[źródło: opracowanie własne]

Z kolei kod w Matlabie inicjalizuje strukturę grafu, reprezentując go za pomocą macierzy sąsiedztwa. Plik wejściowy był otwierany, a następnie wczytywane zostały z niego informacje dotyczące liczby wierzchołków V oraz liczby krawędzi E. Następnie stworzona została macierz sąsiedztwa adjMatrix o wymiarach $V \cdot V$, wypełniona zerami. W pętli, dla każdej krawędzi, wczytywane zostały wierzchołki u i v oraz waga w, które następnie zostały zapisywane do macierzy sąsiedztwa, dokonując aktualizacji odpowiednich pozycji w obu kierunkach, co umożliwiło reprezentację grafu nieskierowanego. Na końcu plik został zamknięty, a czas wykonywania dalszych operacji został zmierzony za pomocą funkcji tic. Fragment kodu przedstawiający implementację struktury grafu w Matlabie przedstawiony został na rysunku 102.

¹⁴ Graf nieskierowany - zbiór wierzchołków połączonych krawędziami bez określonego kierunku.

```

function christofides_tsp()
filename = 'C:\\\\Users\\\\mcmys\\\\OneDrive\\\\Pulpit\\\\magisterka_repo\\\\magisterka-repo\\\\Programy\\\\testy\\\\edges100.in';

fileID = fopen(filename, 'r');
if fileID == -1
    error('Cannot open the file.');
end

V = fscanf(fileID, '%d', 1);
E = fscanf(fileID, '%d', 1);
adjMatrix = zeros(V, V);

for i = 1:E
    u = fscanf(fileID, '%d', 1);
    v = fscanf(fileID, '%d', 1);
    w = fscanf(fileID, '%f', 1);
    adjMatrix(u, v) = w;
    adjMatrix(v, u) = w;
end
fclose(fileID);

tic;
% ... Dalsza część kodu
end

```

Rysunek 102. Implementacja struktury Graph w Matlabie
[źródło: opracowanie własne]

W kolejnym etapie dokonana została implementacja algorytmu Prima, służąca do obliczeni minimalnego drzewa rozpinającego. Na początku zainicjalizowana została liczba wierzchołków V . W kolejnym kroku utworzono nowy graf mst . Zmienna key została wypełniona wartościami maksymalnymi, parent wartościami -1 , a $inMST$ wartościami $false$. Pierwszy element key ustwiono na 0.0 . Następnie, w pętli, dla każdego wierzchołka, została znaleziona najmniejsza wartość key , która nie była jeszcze w MST , oznaczona jako u . Wierzchołek u został dodany do minimalnego drzewa rozpinającego. Kolejna pętla aktualizowała wartości key dla sąsiadów u , jeśli krawędź miała mniejszą wagę niż aktualna wartość key . Po zakończeniu pętli głównej, została obliczona całkowita waga MST poprzez sumowanie wag krawędzi w minimalnym drzewie rozpinającym. Wynikowy graf MST został zwrocony. Fragment kodu implementacji algorytmu Prima przedstawiony został na rysunku 103.

```

// Funkcja obliczająca MST za pomocą algorytmu Prima
Graph primMST(const Graph& graph, double& mstCost) {
    int V = graph.V;
    Graph mst(V);
    std::vector<double> key(V, std::numeric_limits<double>::max());
    std::vector<int> parent(V, -1);
    std::vector<bool> inMST(V, false);

    key[0] = 0.0;

    for (int count = 0; count < V - 1; ++count) {
        double minKey = std::numeric_limits<double>::max();
        int u;

        for (int v = 0; v < V; ++v) {
            if (!inMST[v] && key[v] < minKey) {
                minKey = key[v];
                u = v;
            }
        }

        inMST[u] = true;

        for (int v = 0; v < V; ++v) {
            if (graph.adjMatrix[u][v] && !inMST[v] && graph.adjMatrix[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph.adjMatrix[u][v];
            }
        }
    }

    mstCost = 0.0;
    for (int i = 1; i < V; ++i) {
        mst.addEdge(parent[i], i, graph.adjMatrix[parent[i]][i]);
        mstCost += graph.adjMatrix[parent[i]][i];
    }

    return mst;
}

```

Rysunek 103. Implementacja algorytmu Prima w języku C++
[źródło: opracowanie własne]

W analogiczny sposób algorytm Prima został zaimplementowany w języku Python. Cechą odróżniającą omówiony kod od implementacji C++ użycie list dynamicznych zamiast wektorów, dzięki czemu składnia została uproszczona. Fragment kodu przedstawiający algorytm Prima w Pythonie został przedstawiony na rysunku 104.

```

def prim_mst(graph):
    V = graph.V
    mst = Graph(V)
    key = [float('inf')] * V
    parent = [-1] * V
    in_mst = [False] * V

    key[0] = 0.0

    for _ in range(V - 1):
        min_key = float('inf')
        u = -1

        for v in range(V):
            if not in_mst[v] and key[v] < min_key:
                min_key = key[v]
                u = v

        in_mst[u] = True

        for v in range(V):
            if graph.adjMatrix[u][v] and not in_mst[v] and graph.adjMatrix[u][v] < key[v]:
                parent[v] = u
                key[v] = graph.adjMatrix[u][v]

    mst_cost = 0.0
    for i in range(1, V):
        mst.add_edge(parent[i], i, graph.adjMatrix[parent[i]][i])
        mst_cost += graph.adjMatrix[parent[i]][i]

    return mst, mst_cost

```

Rysunek 104. Implementacja algorytmu Prima w języku Python.
[źródło: opracowanie własne]

Implementacji algorytmu wykonana została również za pomocą programu Matlab. Na początku została zainicjalizowana macierz mst o wymiarach $V \cdot V$ wypełniona zerami. Wektor key uzupełniony został wartościami inf, natomiast wektor parent wartościami -1. Wektor inMST wypełniony został wartościami false. Pierwszy element key ustawiono na 0. W pętli, dla każdego wierzchołka, została znaleziona najmniejsza wartość key, która nie była jeszcze w MST, oznaczona jako u, i dodana do MST. Kolejna pętla aktualizowała wartości key dla sąsiadów u, jeśli krawędź miała mniejszą wagę niż aktualna wartość key. Po zakończeniu pętli głównej, całkowita waga MST została obliczona poprzez sumowanie wag krawędzi w minimalnym drzewie rozpinającym. Wynikowy graf MST oraz jego całkowity koszt zostały zwrócone. Różnica między implementacją w Matlabie a Pythonie polega na użyciu macierzy i wektorów specyficznych dla Matlab, co jest bardziej matematycznie zorientowane i mniej dynamiczne niż listy w Pythonie. Fragment kodu implementacji algorytmu Prima został przedstawiony na rysunku 105.

```

function [mst, mstCost] = primMST(adjMatrix, V)
    mst = zeros(V, V);
    key = inf(1, V);
    parent = -ones(1, V);
    inMST = false(1, V);

    key(1) = 0;

    for count = 1:V-1
        minKey = inf;
        u = -1;

        for v = 1:V
            if ~inMST(v) && key(v) < minKey
                minKey = key(v);
                u = v;
            end
        end

        inMST(u) = true;

        for v = 1:V
            if adjMatrix(u, v) && ~inMST(v) && adjMatrix(u, v) < key(v)
                parent(v) = u;
                key(v) = adjMatrix(u, v);
            end
        end
    end

    mstCost = 0;
    for i = 2:V
        mst(parent(i), i) = adjMatrix(parent(i), i);
        mst(i, parent(i)) = adjMatrix(parent(i), i);
        mstCost = mstCost + adjMatrix(parent(i), i);
    end
end

```

Rysunek 105. Implementacja algorytmu Prima w Matlabie.
[źródło: opracowanie własne]

W kolejnym etapie zaimplementowana została funkcja odnajdująca wierzchołki o nieparzystym stopniu w grafie reprezentowanym przez macierz sąsiedztwa. Na początku zainicjalizowana została pusta lista oddDegreeVertices. Następnie, dla każdego wierzchołka

i, został obliczony stopień wierzchołka poprzez iterację po wszystkich wierzchołkach j oraz zliczanie niezerowych wartości w macierzy sąsiedztwa. Jeśli stopień wierzchołka i był nieparzysty, został dodany do listy oddDegreeVertices. Na końcu lista wierzchołków o nieparzystym stopniu została zwrócona. Fragment kodu odpowiedzialny za implementację omówionej funkcji został przedstawiony na rysunku 106.

```
// Funkcja znajdująca wierzchołki o nieparzystym stopniu
std::vector<int> findOddDegreeVertices(const Graph& graph) {
    std::vector<int> oddDegreeVertices;
    for (int i = 0; i < graph.V; ++i) {
        int degree = 0;
        for (int j = 0; j < graph.V; ++j) {
            if (graph.adjMatrix[i][j] != 0) {
                ++degree;
            }
        }
        if (degree % 2 == 1) {
            oddDegreeVertices.push_back(i);
        }
    }
    return oddDegreeVertices;
}
```

Rysunek 106. Funkcja odnajdująca wierzchołki o nieparzystym stopniu w języku C++.
[źródło: opracowanie własne]

W analogiczny sposób dokonano implementacji funkcji znajdującej wierzchołki posiadające nieparzysty stopień w języku Python. Cechą odróżniającą tą implementację od wykonyanej w języku C++ to wykorzystanie list comprehensions w Pythonie. Zastosowanie list upraszcza oraz skraca kod. Omówiona implementacja przedstawiona została na rysunku 107.

```
def find_odd_degree_vertices(graph):
    odd_degree_vertices = []
    for i in range(graph.V):
        degree = sum(1 for j in range(graph.V) if graph.adjMatrix[i][j] != 0)
        if degree % 2 == 1:
            odd_degree_vertices.append(i)
    return odd_degree_vertices
```

Rysunek 107. Funkcja odnajdująca wierzchołki o nieparzystym stopniu w języku Python.
[źródło: opracowanie własne]

Funkcja znajdująca wierzchołki o nieparzystym stopniu zaimplementowana została również w programie Matlab. Różnica pomiędzy tą, a poprzednimi implementacjami polega na indeksowaniu macierzy rozpoczynającemu się od 1, podczas gdy w Pythonie i C++ rozpoczynane było od wartości 0. W Matlabie do zliczania niezerowych wartości w wierszu macierzy użyto funkcji sum i operatora ~=, co upraszcza obliczenia. Dodawanie elementów do listy odbywa się poprzez konkatenację, co różni się od metody append w Pythonie oraz push_back w C++. Fragment kodu funkcji przedstawiony został na rysunku 108.

```

function oddDegreeVertices = findOddDegreeVertices(mst, V)
    oddDegreeVertices = [];
    for i = 1:V
        degree = sum(mst(i, :) ~= 0);
        if mod(degree, 2) == 1
            oddDegreeVertices = [oddDegreeVertices, i];
        end
    end
end

```

Rysunek 108. Funkcja odnajdująca wierzchołki o nieparzystym stopniu w Matlabie.

[źródło: opracowanie własne]

W kolejnym etapie dokonano implementacji funkcji do znajdowania minimalnego doskonałego skojarzenia w grafie. Na początku zainicjalizowana została pusta lista matching, która przechowywała pary wierzchołków tworzących skojarzenia. Następnie obliczono liczbę wierzchołków o nieparzystym stopniu oraz zainicjalizowano wektor visited jako false dla wszystkich wierzchołków. Dla każdego wierzchołka i z listy wierzchołków o nieparzystym stopniu, w przypadku, gdy wierzchołek ten nie był odwiedzony, został wybrany jako u . W kolejnej pętli, dla każdego wierzchołka j , który nie był odwiedzony i różnił się od i , została sprawdzona waga krawędzi między u a j . Jeśli waga ta była mniejsza niż aktualnie znana minimalna waga, wartości minWeight i minVertex zostały zaktualizowane. Po znalezieniu minimalnej wagi dla u , wierzchołek minVertex został oznaczony jako odwiedzony, a para $(u, \text{oddDegreeVertices}[\text{minVertex}])$ została dodana do listy matching. Na końcu zwrócono listę matching, zawierającą minimalne doskonałe skojarzenie wierzchołków o nieparzystym stopniu. Fragment implementacji omówionej funkcji został przedstawiony na rysunku 109.

```

std::vector<std::pair<int, int>> minWeightMatching(const Graph& graph, const std::vector<int>& oddDegreeVertices) {
    std::vector<std::pair<int, int>> matching;
    int n = oddDegreeVertices.size();
    std::vector<bool> visited(n, false);

    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            int u = oddDegreeVertices[i];
            double minWeight = std::numeric_limits<double>::max();
            int minVertex = -1;

            for (int j = 0; j < n; ++j) {
                if (i != j && !visited[j] && graph.adjMatrix[u][oddDegreeVertices[j]] < minWeight) {
                    minWeight = graph.adjMatrix[u][oddDegreeVertices[j]];
                    minVertex = j;
                }
            }

            visited[minVertex] = true;
            matching.push_back({u, oddDegreeVertices[minVertex]});
        }
    }

    return matching;
}

```

Rysunek 109. Implementacja funkcji minWeightMatching w języku C++.

[źródło: opracowanie własne]

W sposób analogiczny dokonano implementacji funkcji odnajdowania minimalnego doskonałego skojarzenia w grafie za pomocą języka Python. Fragment zaimplementowanego kodu został przedstawiony na rysunku 110.

```

def min_weight_matching(graph, odd_degree_vertices):
    matching = []
    n = len(odd_degree_vertices)
    visited = [False] * n

    for i in range(n):
        if not visited[i]:
            u = odd_degree_vertices[i]
            min_weight = float('inf')
            min_vertex = -1

            for j in range(n):
                if i != j and not visited[j] and graph.adjMatrix[u][odd_degree_vertices[j]] < min_weight:
                    min_weight = graph.adjMatrix[u][odd_degree_vertices[j]]
                    min_vertex = j

            visited[min_vertex] = True
            matching.append((u, odd_degree_vertices[min_vertex]))

    return matching

```

Rysunek 110. Implementacja funkcji minWeightMatching w języku Python.

[źródło: opracowanie własne]

Funkcja minWeightMatching zaimplementowana została również w programie Matlab. Różnica pomiędzy tą, a poprzednimi implementacjami polega na indeksowaniu macierzy rozpoczynającemu się od 1, podczas gdy w Pythonie i C++ rozpoczynane było od wartości 0. W Matlabie do zliczania niezerowych wartości w wierszu macierzy użyto funkcji sum i operatora ~=, co upraszcza obliczenia. Dodawanie elementów do listy odbywa się poprzez konkatenację, co różni się od metody append w Pythonie oraz push_back w C++. Fragment kodu funkcji przedstawiony został na rysunku 111.

```

function matching = minWeightMatching(adjMatrix, oddDegreeVertices)
    n = length(oddDegreeVertices);
    visited = false(1, n);
    matching = [];

    for i = 1:n
        if ~visited(i)
            u = oddDegreeVertices(i);
            minWeight = inf;
            minVertex = -1;

            for j = 1:n
                if i ~= j && ~visited(j) && adjMatrix(u, oddDegreeVertices(j)) < minWeight
                    minWeight = adjMatrix(u, oddDegreeVertices(j));
                    minVertex = j;
                end
            end

            visited(minVertex) = true;
            matching = [matching; u, oddDegreeVertices(minVertex)];
        end
    end
end

```

Rysunek 111. Implementacja funkcji minWeightMatching w Matlabie.

[źródło: opracowanie własne]

W kolejnym kroku zaimplementowana została funkcja realizująca algorytm Christofidesa. Na początku obliczone zostało minimalne drzewo rozpinające za pomocą funkcji primMST, a jego koszt przypisano do mstCost. Następnie, wierzchołki o nieparzystym stopniu zostały znalezione w MST przy użyciu funkcji findOddDegreeVertices. Zostało obliczone minimalne

doskonałe skojarzenie dla tych wierzchołków przy użyciu funkcji minWeightMatching. W kolejnym etapie krawędzie skojarzenia zostały dodane do minimalnego drzewa rozpinającego, aby utworzyć graf eulerowski. Cykl Eulera w powstały grafie został znaleziony za pomocą stosu, iterując przez wierzchołki i oznaczając odwiedzone krawędzie jako 0. Fragment kodu implementacji przedstawiony został na rysunku 112.

```
std::vector<int> christofidesAlgorithm(const Graph& graph, double& tourCost, double& mstCost) {
    Graph mst = primMST(graph, mstCost);

    std::vector<int> oddDegreeVertices = findOddDegreeVertices(mst);

    std::vector<std::pair<int, int>> matching = minWeightMatching(graph, oddDegreeVertices);

    for (const auto& edge : matching) {
        mst.addEdge(edge.first, edge.second, graph.adjMatrix[edge.first][edge.second]);
    }

    std::vector<int> eulerianCycle;
    std::vector<bool> visited(graph.V, false);
    std::vector<int> stack;
    stack.push_back(0);

    while (!stack.empty()) {
        int u = stack.back();
        bool hasUnvisited = false;

        for (int v = 0; v < graph.V; ++v) {
            if (mst.adjMatrix[u][v] != 0) {
                stack.push_back(v);
                mst.adjMatrix[u][v] = 0;
                mst.adjMatrix[v][u] = 0;
                hasUnvisited = true;
                break;
            }
        }

        if (!hasUnvisited) {
            eulerianCycle.push_back(u);
            stack.pop_back();
        }
    }
}
```

Rysunek 112. Fragment implementacji algorytmu Christofidesa w języku C++.
[źródło: opracowanie własne]

W sposób analogiczny przeprowadzona została implementacja w języku Python. Znaczącą różnicą pomiędzy kodami jest użycie w przypadku Pythona list oraz list dwuwymiarowych do reprezentacji grafu. Do obsługi grafu wykorzystana została metoda append oraz bezpośredni dostęp do elementów list. W kontekście obsługi grafu Eurela, w języku Python dokonano skopiowania macierzy sąsiedztwa przed iteracją. Fragment kodu implementacji przedstawiony został na rysunku 113.

```

def christofides_algorithm(graph):
    mst, mst_cost = prim_mst(graph)
    odd_degree_vertices = find_odd_degree_vertices(mst)
    matching = min_weight_matching(graph, odd_degree_vertices)

    for u, v in matching:
        mst.add_edge(u, v, graph.adjMatrix[u][v])

    def find_eulerian_cycle(graph):
        stack = [0]
        eulerian_cycle = []
        adj_matrix_copy = [row[:] for row in graph.adjMatrix]

        while stack:
            u = stack[-1]
            has_unvisited = False

            for v in range(graph.V):
                if adj_matrix_copy[u][v] != 0:
                    stack.append(v)
                    adj_matrix_copy[u][v] = 0
                    adj_matrix_copy[v][u] = 0
                    has_unvisited = True
                    break

            if not has_unvisited:
                eulerian_cycle.append(u)
                stack.pop()

    return eulerian_cycle

```

Rysunek 113. Fragment implementacji algorytmu Christofidesa w języku Python.
[źródło: opracowanie własne]

Implementacja algorytmu Christofidesa w Matlabie wygląda nieco inaczej niż w przypadku poprzednich implementacji. Na początku został uruchomiony stoper za pomocą funkcji tic. Następnie zostało obliczone minimalne drzewo rozpinające za pomocą funkcji primMST, a jego koszt przypisano do mstCost. Wierzchołki o nieparzystym stopniu w MST zostały znalezione przy użyciu funkcji findOddDegreeVertices. Minimalne doskonałe skojarzenie dla tych wierzchołków zostało obliczone za pomocą funkcji minWeightMatching. Krawędzie skojarzenia zostały dodane do minimalnego drzewa rozpinającego, aby utworzyć graf eulerowski. Cykl Eulera w powstałym grafie został znaleziony za pomocą stosu w funkcji findEulerianCycle, gdzie iterowano przez wierzchołki i oznaczano odwiedzone krawędzie jako 0. W odróżnieniu od pozostałych implementacji Matlab używa funkcji macierzowych i operacji wektorowych, co upraszcza operacje na danych i skracą kod. Indeksy z kolei zaczynają się od 1, co różni się od Pythona i C++, gdzie zaczynają się od 0. Fragment implementacji algorytmu Christofidesa przedstawiony został na rysunku 114.

```

function christofides_tsp()
tic;

[mst, mstCost] = primMST(adjMatrix, V);

oddDegreeVertices = findOddDegreeVertices(mst, V);

matching = minWeightMatching(adjMatrix, oddDegreeVertices);

for i = 1:size(matching, 1)
    u = matching(i, 1);
    v = matching(i, 2);
    mst(u, v) = adjMatrix(u, v);
    mst(v, u) = adjMatrix(u, v);
end

eulerianCycle = findEulerianCycle(mst, V);

end

function eulerianCycle = findEulerianCycle(mst, V)
eulerianCycle = [];
stack = [1];
adjMatrix = mst;

while ~isempty(stack)
    u = stack(end);
    hasUnvisited = false;

    for v = 1:V
        if adjMatrix(u, v) ~= 0
            stack = [stack, v];
            adjMatrix(u, v) = 0;
            adjMatrix(v, u) = 0;
            hasUnvisited = true;
            break;
        end
    end

    if ~hasUnvisited
        eulerianCycle = [eulerianCycle, u];
        stack(end) = [];
    end
end
end

```

Rysunek 114. Fragment implementacji algorytmu Christofidesa w Matlabie.
[źródło: opracowanie własne]

Po znalezieniu cyklu Eulera, został utworzony cykl Hamiltona poprzez pominięcie powtarzających się wierzchołków. Na końcu została obliczona suma wag krawędzi trasy, iterując przez kolejne wierzchołki cyklu Hamiltona i sumując wagi krawędzi. Dodano również koszt powrotu do startowego wierzchołka. Funkcja zwraca cykl Hamiltona jako wynik. Fragment kodu implementacji przedstawiony został na rysunku 115.

```

std::vector<int> hamiltonianCycle;
std::vector<bool> visitedHamiltonian(graph.V, false);

for (int v : eulerianCycle) {
    if (!visitedHamiltonian[v]) {
        hamiltonianCycle.push_back(v);
        visitedHamiltonian[v] = true;
    }
}

tourCost = 0;
for (size_t i = 0; i < hamiltonianCycle.size() - 1; ++i) {
    tourCost += graph.adjMatrix[hamiltonianCycle[i]][hamiltonianCycle[i + 1]];
}
tourCost += graph.adjMatrix[hamiltonianCycle.back()][hamiltonianCycle[0]];

return hamiltonianCycle;

```

Rysunek 115. Dalsza część implementacji algorytmu Christofidesa w języku C++.
[źródło: opracowanie własne]

W sposób analogiczny przeprowadzona została implementacja dalszej części algorytmu Christofidesa w języku Python. Fragment kodu odpowiedzialny za utworzenie cyklu Hamiltona został przedstawiony na rysunku 116.

```
eulerian_cycle = find_eulerian_cycle(mst)
hamiltonian_cycle = []
visited = [False] * graph.V

for v in eulerian_cycle:
    if not visited[v]:
        hamiltonian_cycle.append(v)
        visited[v] = True

tour_cost = 0
for i in range(len(hamiltonian_cycle) - 1):
    tour_cost += graph.adjMatrix[hamiltonian_cycle[i]][hamiltonian_cycle[i + 1]]
    tour_cost += graph.adjMatrix[hamiltonian_cycle[-1]][hamiltonian_cycle[0]]

return hamiltonian_cycle, tour_cost, mst_cost
```

Rysunek 116. Dalsza część implementacji algorytmu Christofidesa w języku Python.
[źródło: opracowanie własne]

Dokonana została również dalsza implementacja algorytmu Christofidesa za pomocą programu Matlab. Cykl Hamiltona został utworzony z cyklu Eulera poprzez pominięcie powtarzających się wierzchołków w funkcji eulerianToHamiltonian. Na końcu została obliczona suma wag krawędzi trasy poprzez iterację przez kolejne wierzchołki cyklu Hamiltona i sumowanie wag krawędzi, dodając również koszt powrotu do startowego wierzchołka. Dalsza część implementacji w Matlabie przedstawiona została na rysunku 117.

```
function christofides_tsp()

% Wcześniejszta część kodu
[hamiltonianCycle, tourCost] = eulerianToHamiltonian(eulerianCycle, adjMatrix, V);

duration = toc;

end

%Dalsza część kodu

function [hamiltonianCycle, tourCost] = eulerianToHamiltonian(eulerianCycle, adjMatrix, V)
hamiltonianCycle = [];
visitedHamiltonian = false(1, V);

for v = eulerianCycle
    if ~visitedHamiltonian(v)
        hamiltonianCycle = [hamiltonianCycle, v];
        visitedHamiltonian(v) = true;
    end
end

tourCost = 0;
for i = 1:length(hamiltonianCycle)-1
    tourCost = tourCost + adjMatrix(hamiltonianCycle(i), hamiltonianCycle(i+1));
end
tourCost = tourCost + adjMatrix(hamiltonianCycle(end), hamiltonianCycle(1));
end
```

Rysunek 117. Dalsza część implementacji algorytmu Christofidesa w Matlabie.
[źródło: opracowanie własne]

Kolejnym etapem implementacji programu algorytmu Christofidesa było przedstawienie funkcji głównej programu. Na początku plik wejściowy został otwarty za pomocą std::ifstream. Dzięki zaimplementowanej obsłudze błędów, w przypadku problemów związanych z otwarciem pliku, wypisano stosowny komunikat i zakończono program. Następnie zostały wczytane liczba wierzchołków V i krawędzi E, a także utworzono graf graph o V wierzchołkach. Krawędzie zostały wczytane w pętli i dodane do grafu, przy czym indeksy zostały przekształcone na bazę 0. Czas wykonania algorytmu został zmierzony za pomocą high_resolution_clock. Obliczono trasę przy użyciu funkcji christofidesAlgorithm, której wynikiem była trasa tour, koszt trasy tourCost oraz koszt MST mstCost. Na końcu wyniki zostały wyświetlane: trasa, suma wag krawędzi trasy oraz suma wag krawędzi w minimalnym drzewie rozpinającym. Czas działania programu został zmierzony i wyświetlony z trzema miejscami po przecinku. Kod implementacji funkcji głównej programu przedstawiony został na rysunku 118.

```

int main() {
    std::ifstream inputFile("C:\\\\Users\\\\mcmys\\\\OneDrive\\\\Pulpit\\\\magisterka repo\\\\magisterka-repo\\\\Programy\\\\testy\\\\edges100.in");
    if (!inputFile) {
        std::cerr << "Nie można otworzyć pliku." << std::endl;
        return 1;
    }

    int V, E;
    inputFile >> V >> E;
    Graph graph(V);

    int u, v;
    double w;
    while (inputFile >> u >> v >> w) {
        graph.addEdge(u - 1, v - 1, w);
    }

    auto start = high_resolution_clock::now();

    double tourCost, mstCost;
    std::vector<int> tour = christofidesAlgorithm(graph, tourCost, mstCost);

    auto stop = high_resolution_clock::now();
    duration<double> duration = stop - start;

    std::cout << "Znaleziona trasa: ";
    for (int v : tour) {
        std::cout << (v + 1) << " ";
    }
    std::cout << (tour[0] + 1) << std::endl;

    std::cout << "Suma wag krawędzi trasy: " << tourCost << std::endl;
    std::cout << "Suma wag krawędzi w MST: " << mstCost << std::endl;
    std::cout << std::fixed << std::setprecision(3);
    std::cout << "Czas działania programu: " << duration.count() << " sekund" << std::endl;

    saveResults(tour, tourCost, mstCost, duration.count());
}

return 0;
}

```

Rysunek 118. Funkcja główna programu w języku C++.
[źródło: opracowanie własne]

W sposób analogiczny dokonana została implementacja funkcji głównej programu w języku Python. Fragment kodu funkcji main() został przedstawiony na rysunku 119.

```

def main():
    with open("C:\\\\Users\\\\mcmys\\\\OneDrive\\\\Pulpit\\\\magisterka_repo\\\\magisterka-repo\\\\Programy\\\\testy\\\\edges100.in") as input
        V, E = map(int, input_file.readline().split())
        graph = Graph(V)

        for line in input_file:
            u, v, w = map(float, line.split())
            graph.add_edge(int(u) - 1, int(v) - 1, w)

    start = time.time()

    tour, tour_cost, mst_cost = christofides_algorithm(graph)

    duration = time.time() - start

    print("Znaleziona trasa: ", ".join(str(v + 1) for v in tour), tour[0] + 1)
    print("Suma wag krawędzi trasy:", tour_cost)
    print("Suma wag krawędzi w MST:", mst_cost)
    print("Czas działania programu:", f"{duration:.3f} sekund")

    save_results(tour, tour_cost, mst_cost, duration)

if __name__ == "__main__":
    main()

```

Rysunek 119. Funkcja główna programu w języku Python.
[źródło: opracowanie własne]

Implementacja funkcji głównej programu wykonana w Matlabie rozpoczyna działanie od pomiaru czasu wykonania algorytmu Christofidesa. Na początku został uruchomiony stoper za pomocą funkcji tic, a następnie zatrzymany za pomocą funkcji toc, co pozwoliło zmierzyć czas działania algorytmu i przypisać go do zmiennej duration. Następnie zostały wyświetlane wyniki: znalezioną trasę hamiltonianCycle, sumę wag krawędzi trasy tourCost, sumę wag krawędzi w MST mstCost oraz czas działania programu duration. Trasa została wyświetlona w formacie, gdzie wierzchołki są wypisywane w jednej linii, a na końcu powrót do początkowego wierzchołka. Wyniki zostały zapisane do pliku tekstowego za pomocą funkcji saveResults oraz do pliku CSV za pomocą funkcji saveResultsToCSV. Fragment kodu odpowiedzialny za omówioną funkcję przedstawiony został na rysunku 120.

```

function christofides_tsp()

% Wcześniejszia część kodu
tic;
% Późniejsza część kodu

duration = toc;

fprintf('Znaleziona trasa: ');
for i = 1:length(hamiltonianCycle)
    fprintf('%d ', hamiltonianCycle(i));
end
fprintf('%d\n', hamiltonianCycle(1));
fprintf('Suma wag krawędzi trasy: %.2f\n', tourCost);
fprintf('Suma wag krawędzi w MST: %.2f\n', mstCost);
fprintf('Czas działania programu: %.6f sekund\n', duration);

saveResults(hamiltonianCycle, tourCost, mstCost, duration);
saveResultsToCSV(hamiltonianCycle, tourCost, mstCost, duration);
end

```

Rysunek 120. Funkcja główna programu w Matlabie.
[źródło: opracowanie własne]

W końcowej fazie implementacji dokonano zapisu wyników działania programu do pliku CSV. Do pliku zostały zapisane: lista wierzchołków path (przekształcona na bazę 1), koszt trasy totalCost, koszt MST mstCost oraz czas wykonania duration. Wartości również zostały zapisane z trzema miejscami po przecinku. Na końcu oba pliki zostały zamknięte. Fragment kodu implementacji odpowiedzialny za zapisanie wyników do plików CSV, został zaprezentowany na rysunku 121.

```
void saveResults(const std::vector<int>& path, double totalCost, double mstCost, double duration) {
    std::ofstream csvFile("results.csv");
    csvFile << "Vertex\n";
    for (int v : path) {
        csvFile << v + 1 << "\n";
    }
    csvFile << "Total cost of the tour," << totalCost << "\n";
    csvFile << "Total cost of the MST," << mstCost << "\n";
    csvFile << std::fixed << std::setprecision(3);
    csvFile << "Execution time (seconds)," << duration << "\n";
    csvFile.close();
}
```

Rysunek 121. Funkcja saveResults() w języku C++.
[źródło: opracowanie własne]

W sposób analogiczny zaimplementowana została funkcja zapisu wyników do plików tekstowych oraz CSV w języku Python. Fragment implementacji przedstawiający ten proces został przedstawiony na rysunku 122.

```
def save_results(path, total_cost, mst_cost, duration):
    with open("results.csv", "w") as csv_file:
        csv_file.write("Vertex\n")
        for v in path:
            csv_file.write(f"{v + 1}\n")
        csv_file.write(f"Total cost of the tour,{total_cost}\n")
        csv_file.write(f"Total cost of the MST,{mst_cost}\n")
        csv_file.write(f"Execution time (seconds),{duration:.3f}\n")
```

Rysunek 122. Funkcja saveResults() w języku Python.
[źródło: opracowanie własne]

Zapis uzyskanych wyników do pliku CSV zaimplementowany został również za pomocą programu Matlab. Fragment kodu odpowiedzialny za ten proces przedstawiony został na rysunku 123.

```
function saveResultsToCSV(path, totalCost, mstCost, duration)
fileID = fopen('results.csv', 'w');
fprintf(fileID, 'Vertex\n');
for i = 1:length(path)
    fprintf(fileID, '%d\n', path(i));
end
fprintf(fileID, 'Total cost of the tour,%2f\n', totalCost);
fprintf(fileID, 'Total cost of the MST,%2f\n', mstCost);
fprintf(fileID, 'Execution time (seconds),%6f\n', duration);
fclose(fileID);
end
```

Rysunek 123. Funkcja saveResultsToCSV zaimplementowana w Matlabie.
[źródło: opracowanie własne]

W celu potwierdzenia poprawności działania opracowanych algorytmów Christofidesa w języku C++, Python oraz Matlab przeprowadzono badanie na wygenerowanych podczas przedstawiania działania generatora liczb losowych danych wejściowych. Efekt działania przedstawiony został na rysunku 124.

The screenshot shows three separate command-line windows side-by-side, each displaying the output of a Christofides algorithm run on a 100-city TSP instance. The windows are labeled 'Python', 'C++', and 'Matlab' from top to bottom.

Python window output:

```
C:\Windows\System32\cmd.exe
C:\Users\mcmys\OneDrive\Pulpit\magisterka repo\magisterka-repo\Programy\python\chris>python christofides.py
Znaleziona trasa: 30 18 58 93 70 40 55 41 6 84 37 52 38 33 34 28 29 66 27 35 94 48 23 4 51 77 76 82 53 11 47 72 59 62 16 64 39 5 99 88 13 22 57 15 100 98 67 36 79 54 50 44 74 91 80 86 89 46 85 3 9 60 90
8 14 49 83 24 78 81 1 96 32 42 97 61 73 26 45 31 92 2 68 19 17 12 25 71 63 65 95 56 75 10 21 87 43 7 69 20 30
Suma wag krawędzi trasy: 16831.0
Suma wag krawędzi w MST: 16807.0
Czas działania programu: 0.003 sekund
```

C++ window output:

```
C:\Users\mcmys\OneDrive\Pulpit\magisterka repo\magisterka-repo\Programy\c++\christofides\x64\Debug\christofides.exe (proces 33988) zakończono z kodem 0.
Command Window
>>> christofides_tsp
Znaleziona trasa: 30 18 58 93 70 40 55 41 6 84 37 52 38 33 34 28 29 66 27 35 94 48 23 4 51 77 76 82 53 11 47 72 59 62 16 64 39 5 99 88 13 22 57 15 100 98 67 36 79 54 50 44 74 91 80 86 89 46 85 3 9 60 9
Suma wag krawędzi trasy: 16831.00
Suma wag krawędzi w MST: 1007.00
Czas działania programu: 0.010053 sekund
f_ >>
```

Matlab window output:

```
>> christofides_tsp
Znaleziona trasa: 30 18 58 93 70 40 55 41 6 84 37 52 38 33 34 28 29 66 27 35 94 48 23 4 51 77 76 82 53 11 47 72 59 62 16 64 39 5 99 88 13 22 57 15 100 98 67 36 79 54 50 44 74 91 80 86 89 46 85 3 9 60 90
Suma wag krawędzi trasy: 16831.00
Suma wag krawędzi w MST: 1007.00
Czas działania programu: 0.010053 sekund
```

Rysunek 124. Wyniki działania opracowanych programów
[źródło: opracowanie własne]

Rysunek 124 przedstawia wyniki uzyskane w wyniku przeprowadzenia badania na danych wejściowych dla 100 wierzchołków. Trasa znaleziona przez wszystkie implementacje jest identyczna, co świadczy o poprawności działania algorytmu w każdym z języków. Opracowane programy okazały się zgodne również w kwestii wyliczenia sumy wag krawędzi trasy (16831) oraz sumy wag krawędzi w minimalnym drzewie rozpinającym (1007), co świadczy o spójności wyników. Jedyną różnicą pomiędzy programami jest czas działania programu. Najkrótszy czas uzyskany został w algorytmie zaimplementowanym w języku Pythonie i wyniósł 0.003 sekundy. Na pozycji drugiej znajduje się wynik uzyskany przez kod zaimplementowany w języku C++ (0.008 sekund). Ostatnie miejsce z czasem 0.010053 sekund uzyskał program opracowany Matlabie.

7. Badanie porównawcze algorytmów

Rozdział siódmy skupia swoją uwagę na porównaniu zaimplementowanych algorytmów. W celu przeprowadzenia badań za pomocą generatora liczb losowych wygenerowane zostało sześć plików zawierające dane wejściowe. Dane wejściowe utworzone zostały dla 20, 50, 100, 300, 500 oraz 100 wierzchołków.

Do przeprowadzenia badań wykorzystane zostały algorytmy, których implementacja przedstawiona została w rozdziale szóstym tj.:

- algorytm selekcji krawędzi;
- algorytm dwuoptymalny;
- algorytm trójoptymalny;
- algorytm Christofidesa.

Każdy z wymienionych powyżej algorytmów został zaimplementowany w różnych językach programowania, takich jak C++ i Python. Dokonując przeglądu literatury, w tym także pozycji obcojęzycznych, podjęto decyzję o implementacji wyselekcjonowanych algorytmów również w środowisku obliczeń naukowych Matlab. Stwierdzono również brak informacji na temat rozwiązania problemu komiwojażera za pomocą algorytmu selekcji krawędzi, dlatego zdecydowano się na jego implementację we wszystkich wymienionych środowiskach programistycznych. Eksperyment przeprowadzono na trzech różnych urządzeniach komputerowych:

- Laptop wyposażony w procesor Intel Core i7-8750H, 6 rdzeni, 12 wątków, bazowa częstotliwość taktowania 2.20GHz, częstotliwość taktowania turbo 4,10 GHz, pamięć podręczna Cache 9 MB, 16 GB pamięci RAM, system operacyjny Windows 11 Home;
- Komputer wyposażony w procesor Intel Core i5-12400F, 6 rdzeni, 12 wątków, bazowa częstotliwość taktowania 2.50GHz, częstotliwość taktowania turbo 4,40 GHz, pamięć podręczna Cache 18 MB, 32 GB pamięci RAM, system operacyjny Windows 10 Pro;
- Komputer wyposażony w procesor AMD Ryzen 5 5600, 6 rdzeni, 12 wątków, bazowa częstotliwość taktowania 3.50GHz, częstotliwość taktowania turbo 4,40 GHz, pamięć podręczna Cache 35 MB, 16 GB pamięci RAM, system operacyjny Windows 10 Pro.

Badania rozpoczęto od utworzenia, za pomocą generatora grafów losowych danych wejściowych dla: 20, 50, 100, 300, 500, 1000 wierzchołków oraz umieszczenia ich w miejscu, do którego każdy zaimplementowanych programów, w różnych językach miał swobodny dostęp. Następnie zaopatrzono stanowiska komputerowe w odpowiednie oprogramowanie, za pomocą którego przeprowadzone zostały testy.

Analizę porównawczą rozpoczęto od przeprowadzenia badań na programach opracowanych w języku C++ na laptopie o parametrach sprzętowych Intel Core i7-8750H, 6 rdzeni, 12 wątków, bazowa częstotliwość taktowania 2.20GHz, częstotliwość taktowania turbo 4,10 GHz, pamięć podręczna Cache 9 MB, 16 GB pamięci RAM, system operacyjny Windows 11 Home. Uzyskane dane przedstawione zostały w tabeli 8.

Tabela 8.
Wyniki analizy programów w języku C++ na laptopie.

Intel Core i7-8750H						
Algorytm		Ilość wierzchołków				
		20	50	100	300	1000
dwuoptymalny	Całkowita waga trasy	2170	2838	3313	4837	6408
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722
	Czas [s]	0,05	0,06	0,11	0,67	1,99
trójoptymalny	Całkowita waga trasy	1773	2106	2191	2493	2963
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722
	Czas [s]	0,01	0,05	0,79	72,20	544,39
Christofidesa	Całkowita waga trasy	4039	11201	18123	48301	80102
	Suma wszystkich wag krawędzi	970	1251	1097	1196	1395
	Czas [s]	0,00038	0,0022	0,0098	0,098	0,30
selekacji krawędzi	Całkowita waga trasy	2464	3870	4155	4190	4779
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722
	Czas [s]	0.01	0.02	0.08	0.78	1.82

źródło: opracowanie własne

Kontynuując analizę algorytmów opracowanych w języku C++, przeprowadzono badania na drugim stanowisku komputerowym o parametrach: procesor Intel Core i5-12400F, 6 rdzeni, 12 wątków, bazowa częstotliwość taktowania 2.50GHz, częstotliwość taktowania turbo 4,40 GHz, pamięć podręczna Cache 18 MB, 32 GB pamięci RAM, system operacyjny Windows 10 Pro. Wyniki przeprowadzonej analizy przedstawione zostały w tabeli 9.

Tabela 9.

Wyniki analizy programów w języku C++ na komputerze z procesorem Intel.

Intel Core i5-12400F						
Algorytm		Ilość wierzchołków				
		20	50	100	300	1000
dwooptymalny	Calkowita waga trasy	2170	2838	3313	4837	6408
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722
	Czas [s]	0.01	0.02	0.04	0.29	0.91
trójoptymalny	Całkowita waga trasy	1773	2106	2191	2493	2963
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722
	Czas [s]	0.002	0.024	0.32	29.79	234.71
Christofidesa	Całkowita waga trasy	4039	11201	18123	48301	80102
	Suma wszystkich wag krawędzi	970	1251	1097	1196	1395
	Czas [s]	0.00026	0.001	0.005	0.051	0.12
selekcji krawędzi	Całkowita waga trasy	2464	3870	4155	4190	4779
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722
	Czas [s]	0.002	0.009	0.036	0.30	0.86

źródło: opracowanie własne

Analiza porównawcza algorytmów opracowanych w języku C++ przeprowadzona została również na trzecim stanowisku komputerowym. Dla urozmaicenia wyników dokonano wyboru stanowiska, które posiadało procesor firmy AMD. Parametry sprzętowe trzeciego komputera wyglądały następująco: procesor AMD Ryzen 5 5600, 6 rdzeni, 12 wątków, bazowa

częstotliwość taktowania 3.50GHz, częstotliwość taktowania turbo 4,40 GHz, pamięć podręczna Cache 35 MB, 16 GB pamięci RAM, system operacyjny Windows 10 Pro. Wyniki analizy przedstawione zostały w tabeli 10.

Tabela 10.

Wyniki analizy programów w języku C++ na komputerze z procesorem AMD.

		AMD Ryzen 5 5600					
Algorytm		Ilość wierzchołków					
		20	50	100	300	500	1000
dwooptymalny	Całkowita waga trasy	2170	2838	3313	4837	6408	9186
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.01	0.02	0.04	0.30	0.97	5.24
trójoptymalny	Całkowita waga trasy	1773	2106	2191	2493	2963	3569
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.002	0.028	0.40	40.21	390.24	10132,52
Christofidesa	Całkowita waga trasy	4039	11201	18123	48301	80102	146242
	Suma wszystkich wag krawędzi	970	1251	1097	1196	1395	1729
	Czas [s]	0.00032	0.0014	0.0058	0.049	0.16	0.61
selekcji krawędzi	Całkowita waga trasy	2464	3870	4155	4190	4779	5906
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.003	0.01	0.04	0.31	0.90	3.36

źródło: opracowanie własne

W kolejnym etapie badań przeprowadzono analizę porównawczą zaimplementowanych algorytmów w języku Python. Rozpoczęto od przeprowadzenia testów na laptopie wyposażonym w procesor Intel Core i7-8750H, 6 rdzeni, 12 wątków, bazowa częstotliwość taktowania 2.20GHz, częstotliwość taktowania turbo 4,10 GHz, pamięć podręczna Cache 9 MB, 16 GB pamięci RAM, system operacyjny Windows 11 Home. Wyniki uzyskane w wyniku przeprowadzonej analizy, zostały przedstawione w tabeli 11.

Tabela 11.

Wyniki analizy programów w języku Python na laptopie.

Intel Core i7-8750H							
Algorytm		Ilość wierzchołków					
		20	50	100	300	500	1000
dwooptymalny	Całkowita waga trasy	2170	2838	3313	4837	6408	9186
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0,0051	0,052	0,46	13,39	71,35	601,74
trójoptymalny	Całkowita waga trasy	1773	2106	2191	2493	2963	3569
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0,078	4,31	79,43	8334,59	62843,07	1631710,31
Christofidesa	Całkowita waga trasy	4039	11201	18123	48301	80102	146242
	Suma wszystkich wag krawędzi	970	1251	1097	1196	1395	1729
	Czas [s]	0,001	0,0009	0,0035	0,032	0,097	0.46
selekcji krawędzi	Calkowita waga trasy	2464	3870	4155	4190	4779	5906
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0,002	0,015	0,05	0,21	0,62	2,25

źródło: opracowanie własne

W analogiczny sposób analizę statystyczną przeprowadzono dla algorytmów zaimplementowanych w języku Python, za pomocą komputera następujących parametrach: procesor Intel Core i5-12400F, 6 rdzeni, 12 wątków, bazowa częstotliwość taktowania 2.50GHz, częstotliwość taktowania turbo 4,40 GHz, pamięć podrzczna Cache 18 MB, 32 GB pamięci RAM, system operacyjny Windows 10 Pro.

Tabela 12.

Wyniki analizy programów w języku Python na komputerze z procesorem Intel.

Intel Core i5-12400F							
Algorytm		Ilość wierzchołków					
		20	50	100	300	500	1000
dwooptymalny	Całkowita waga trasy	2170	2838	3313	4837	6408	9186
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0,003	0,03	0,027	7,63	71,35	601,74
trójoptymalny	Całkowita waga trasy	1773	2106	2191	2493	2493	2963
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	22323865	61917722
	Czas [s]	0.049	2.44	42.03	4410.21	34747,24	902206,98
Christofidesa	Całkowita waga trasy	4039	11201	18123	48301	80102	146242
	Suma wszystkich wag krawędzi	970	1251	1097	1196	1395	1729
	Czas [s]	0.000997	0.000998	0.002	0.016	0.043	0.20
selekcji krawędzi	Całkowita waga trasy	2464	3870	4155	4190	4779	5906
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.001	0,003	0,01	0.08	0.23	1.00

źródło: opracowanie własne

Kontynuując analizę porównawczą dla algorytmów zaimplementowanych w języku Python przeprowadzono badanie na komputerze wyposażonym w procesor AMD Ryzen 5 5600, 6 rdzeni, 12 wątków, bazowa częstotliwość taktowania 3.50GHz, częstotliwość takto-wania turbo 4,40 GHz, pamięć podrzczna Cache 35 MB, 16 GB pamięci RAM, system operacyjny Windows 10 Pro. Wyniki przeprowadzonej analizy zostały przedstawione w tabeli 13.

Tabela 13.

Wyniki analizy programów w języku Python na komputerze z procesorem AMD.

AMD Ryzen 5 5600							
Algorytm		Ilość wierzchołków					
		20	50	100	300	500	1000
dwuoptymalny	Całkowita waga trasy	2170	2838	3313	4837	6408	9186
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0,06	0,09	0,35	8,80	44.30	386,40
trójoptymalny	Całkowita waga trasy	1773	2106	2191	2493	2963	3569
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.09	2.75	45.57	4781,66	46406,27	1204931,32
Christofidesa	Całkowita waga trasy	4039	11201	18123	48301	80102	146242
	Suma wszystkich wag krawędzi	970	1251	1097	1196	1395	1729
	Czas [s]	0.0009992	0.0009996	0.002	0.02	0.05	0.22
selekcji krawędzi	Calkowita waga trasy	2464	3870	4155	4190	4779	5906
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.02	0,04	0,31	0.51	0.97	2.52

źródło: opracowanie własne

Opracowując wyselekcjonowane algorytmy rozwiązujące problem komiwojażera przeprowadzone zostało doświadczenie, jakim było podjęcie próby ich implementacji za pomocą programu Matlab. Eksperyment zakończył się sukcesem, w wyniku czego dokonano analizy porównawczej opracowanych skryptów. Na początku testy przeprowadzony zostały na laptopie z procesorem Intel Core i7-8750H. Wyniki przeprowadzonej analizy przedstawione zostały w tabeli 14.

Tabela 14.

Wyniki analizy skryptów w Matlabie na laptopie.

Intel Core i7-8750H							
Algorytm		Ilość wierzchołków					
		20	50	100	300	500	1000
dwooptymalny	Całkowita waga trasy	2170	2838	3313	4837	6408	9186
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0,012	0,025	0,041	0,19	0,64	6,77
trójoptymalny	Całkowita waga trasy	1773	2106	2191	2493	2963	3569
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.06	2.89	49.22	6381,18	48144,26	1249279,24
Christofidesa	Całkowita waga trasy	4039	11201	18123	48301	80102	146242
	Suma wszystkich wag krawędzi	970	1251	1097	1196	1395	1729
	Czas [s]	0.005	0.015	0.024	0.043	0.068	0.13
selekcji krawędzi	Całkowita waga trasy	2464	3870	4155	4190	4779	5906
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.06	0.03	0.05	0.26	0.52	2.17

źródło: opracowanie własne

Badanie porównawcze skryptów opracowanych za pomocą programu Matlab zostało wykonane również na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Wyniki testów przedstawione zostały w tabeli 15.

Tabela 15.

Wyniki analizy skryptów w Matlabie na komputerze z procesorem Intel.

Intel Core i5-12400F							
Algorytm		Ilość wierzchołków					
		20	50	100	300	500	1000
dwooptymalny	Całkowita waga trasy	2170	2838	3313	4837	6408	9186
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0,008	0,0016	0,027	0,39	1.97	29.28
trójoptymalny	Całkowita waga trasy	1773	2106	2191	2493	2963	3569
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.06	1.76	30.40	3100,77	24430,39	634331,47
Christofidesa	Całkowita waga trasy	4039	11201	18123	48301	80102	146242
	Suma wszystkich wag krawędzi	970	1251	1097	1196	1395	1729
	Czas [s]	0.004	0.014	0.025	0.036	0.58	0.08
selekcji krawędzi	Calkowita waga trasy	2464	3870	4155	4190	4779	5906
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.03	0.01	0.04	0.13	0.26	0.96

źródło: opracowanie własne

Na sam koniec, w analogiczny sposób przeprowadzono badanie porównawcze algorytmów opracowanych w programie Matlab, na komputerze stacjonarnym wyposażonym w procesor firmy AMD. W tabeli 16 przedstawione zostały uzyskane rezultaty wykonanych testów.

Tabela 16.

Wyniki analizy skryptów w Matlabie na komputerze z procesorem AMD.

AMD Ryzen 5 5600							
Algorytm		Ilość wierzchołków					
		20	50	100	300	500	1000
dwooptymalny	Całkowita waga trasy	2170	2838	3313	4837	6408	9186
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0,025	0,011	0,026	0,14	0,48	3,76
trójoptymalny	Całkowita waga trasy	1773	2106	2191	2493	2963	3569
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0,071	2,27	38,64	3941,24	38249,93	993153,24
Christofidesa	Całkowita waga trasy	4039	11201	18123	48301	80102	146242
	Suma wszystkich wag krawędzi	970	1251	1097	1196	1395	1729
	Czas [s]	0.006	0.013	0.029	0.041	0.062	0.11
selekcji krawędzi	Całkowita waga trasy	2464	3870	4155	4190	4779	5906
	Suma wszystkich wag krawędzi	100268	615568	2458999	22323865	61917722	248352320
	Czas [s]	0.02	0.015	0.023	0.12	0.30	1.13

źródło: opracowanie własne

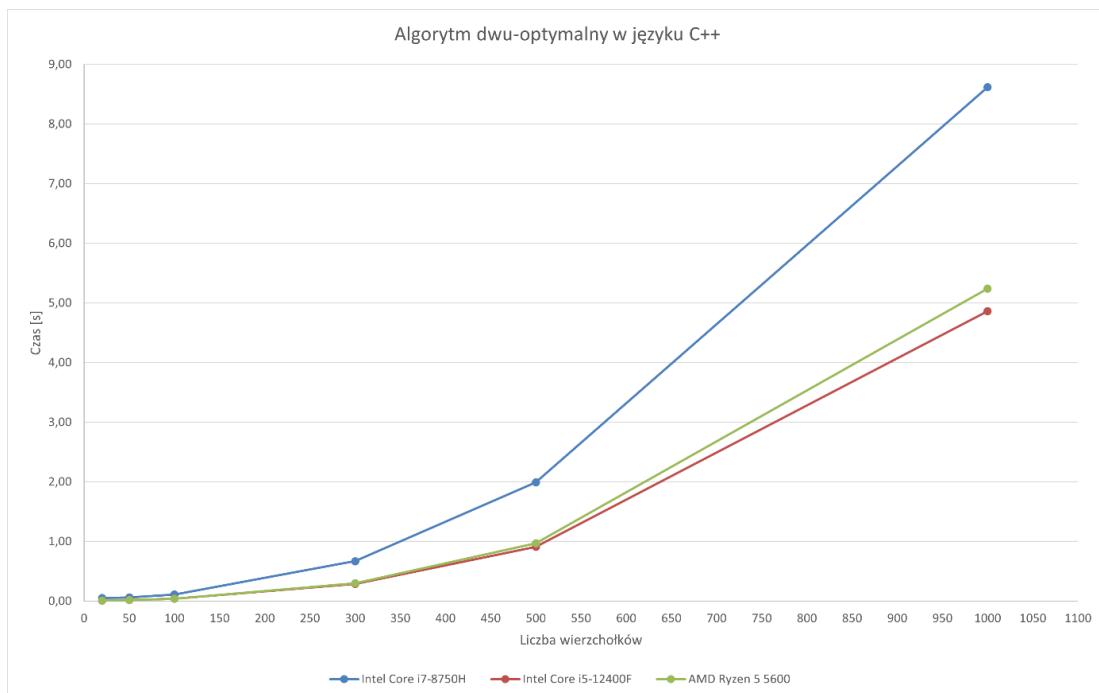
Wyniki oraz wnioski z badań wraz z wykresami przedstawione zostały w rozdziale ósmym.

Pola oznaczone kolorem żółtym w algorytmie trój-optymalnym zawierają szacunkowe czasy wyliczone na podstawie proporcji, ponieważ ciągła praca wszystkich stanowisk komputerowych dla implementacji w Pythonie i Matlabie przekraczała by 1900 godzin.

8. Wyniki badań

Rozdział ósmy skupia swoją uwagę na przedstawieniu wyników przeprowadzonych na potrzeby pracy dyplomowej badań. W ramach analizy porównawczej zaimplementowanych wyselekcjonowanych algorytmów, które przedstawione zostały w rozdziale szóstym oraz przeprowadzonych na nich testach, które zapisane zostały w tabelach 8, 9, 10, 11, 12, 13, 14, 15 oraz 16, które przedstawione zostały w rozdziale siódmym.

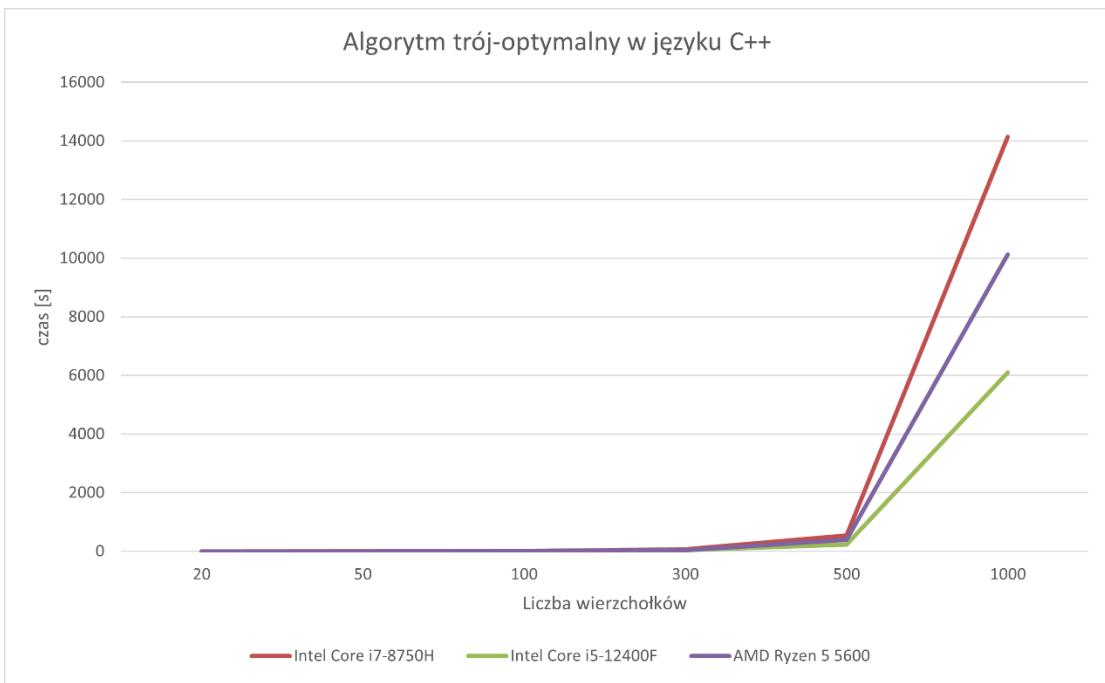
Badania rozpoczęto od wygenerowania danych wejściowych dla 20, 50, 100, 300, 500 oraz 1000 wierzchołków za pomocą wcześniej zaimplementowanego generatora grafów losowych. Wygenerowane dane zostały wraz z wymaganym oprogramowaniem zostały przeniesione na stanowiska komputerowe, na których zadecydowano wykonać testy. Dla urozmaicenia wyników badania przeprowadzone zostały na laptopie oraz dwóch komputerach stacjonarnych, wyposażonych w procesory dwóch różnych firm. Celem badania było sprawdzenie wpływu zasobów sprzętowych oraz ilości wierzchołków na czas rozwiązania problemu komiwojażera. Na początku rozpoczęto przeprowadzanie testów na algorytmach zaimplementowanych w języku C++. Badanie przeprowadzone zostało na trzech różnych stanowiskach pracy. Dane uzyskane w wyniku badań przedstawione zostały w tabelach numer 8, 9 oraz 10. W pierwszym etapie przetestowany został algorytm dwu-optymalny. Wyniki uzyskane w trakcie analizy przedstawione zostały na rysunku 125.



Rysunek 125. Wykres uzyskanych wyników dla algorytmu dwu-optymalnego w języku C++
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 125, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas osiągnęły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku laptopa posiadającego procesor Intel Core i7-8750H. W celu zbadania złożoności obliczeniowej algorytmu przeprowadzona została analiza teoretyczna, która rozpoczęła się od odczytu danych wejściowych z pliku łącznie z liczbą wierzchołków N oraz krawędzi M . Złożoność tego procesu wyniosła $O(M)$. Następnie algorytm zainicjował macierz sąsiedztwa W , której rozmiar wyniósł $N \cdot N$. Proces ten zajął $O(N^2)$ czasu. W kolejnym etapie dokonano kalkulacji głównej pętli zaimplementowanego kodu. Algorytm optymalizował trasę, dokonując iteracji do momentu, aż do momentu, w którym nie istniały dalsze ulepszenia. W każdej z iteracji dokonywał sprawdzenia wszystkich par krawędzi, czego powodem była podwojona pętla o złożoności wynoszącej $O(N^2)$, przy czym każde ze sprawdzeń stanowiło operację stałą $O(1)$. Jeżeli odnalezione zostało ulepszenie, wtedy algorytm dokonywał aktualizacji ścieżki. Proces ten w najgorszym wypadku zajmował $O(N)$. Zewnętrzna pętla kontynuowała do momentu, w którym nie było możliwości odnalezienia dalszych ulepszeń. Zjawisko to trwało do $O(N^2)$ iteracji. Każde powtórzenie zewnętrznej pętli osiągnęło złożoność $O(N^2)$, ze względu na podwójną pętle wewnętrzną. W rezultacie łączna złożoność obliczeniowa algorytmu dwu-optymalnego wyniosła $O(N^4)$. Na podstawie danych otrzymanych podczas przeprowadzania badań na różnych procesorach potwierdziła złożoność algorytmu. Przykładowo wynik dla 1000 wierzchołków dla Intel Core i7-8750H wynosi 8,62 sekundy. Z kolei dla Intel Core i5-12400F czas wyniósł 4,86, natomiast dla AMD Ryzen 5 5600 5,24 sekundy. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, przedstawiony na rysunku 124 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(N^4)$.

W kolejnym etapie przeprowadzano analizę porównawczą dla algorytmu trój-optymalnego zaimplementowanego w języku C++. Badanie przeprowadzone zostało na trzech stanowiskach komputerowych. Dane uzyskane w wyniku badań przedstawione zostały w tabelach numer 8, 9 oraz 10, natomiast w graficzny sposób przedstawione zostały na rysunku 126.

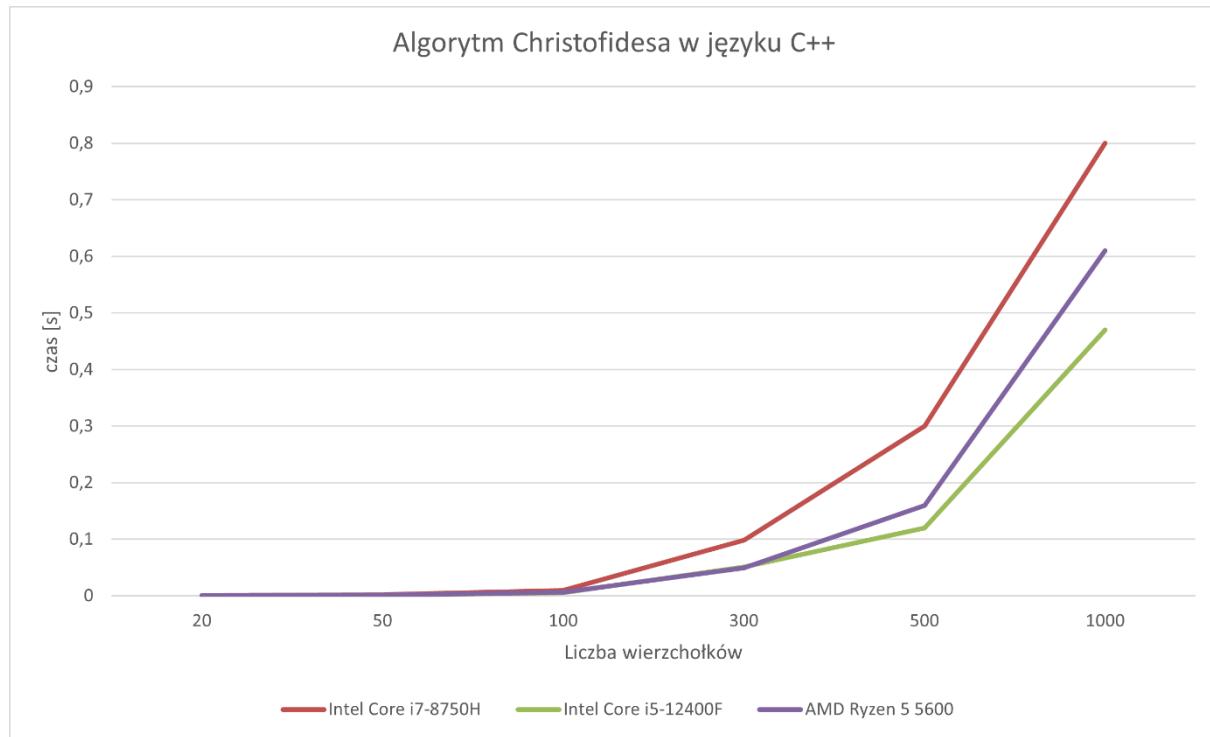


Rysunek 126. Wykres uzyskanych wyników dla algorytmu trój-optymalnego w języku C++
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 126, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas osiągnęły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku laptopa posiadającego procesor Intel Core i7-8750H. W celu zbadania złożoności obliczeniowej algorytmu przeprowadzona została analiza teoretyczna, która rozpoczęła się od odczytu danych wejściowych z pliku, łącznie z liczbą wierzchołków N oraz krawędzi M . Złożoność tego procesu wyniosła $O(M)$. Następnie algorytm zainicjował macierz sąsiedztwa W , w której rozmiar wyniósł $N \cdot N$. Proces ten zajął $O(N^2)$ czasu. W kolejnym etapie algorytm zainicjował tablicę Ptr oraz Route oraz obliczył początkową wagę trasy Tweight . Proces ten miał złożoność $O(N)$. Główna część algorytmu obejmowała iteracyjną optymalizację trasy przy użyciu podejścia 3-opt, które polegało na sprawdzeniu wszystkich możliwych trójków krawędzi w celu znalezienia najlepszej wymiany. Zewnętrzna pętla kontynuowała, dopóki znajdowane były ulepszenia, co mogło trwać do $O(N^3)$ iteracji, ze względu na złożoność sprawdzania wszystkich możliwych trójków krawędzi. W każdej iteracji, sprawdzanie wszystkich trójków krawędzi (dla I, J i K) miało złożoność $O(N^3)$. W najgorszym przypadku, każda iteracja mogła wymagać aktualizacji trasy, co zajmowało $O(N)$ czasu. W rezultacie, łączna złożoność obliczeniowa algorytmu wyniosła $O(N^4)$. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów pokazuje, że czas

wykonania rośnie wykładowczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(N^4)$

W kolejnym etapie przeprowadzono analizę porównawczą dla algorytmu Christofidesa zaimplementowanego w języku C++. Badanie przeprowadzone zostało na trzech stanowiskach komputerowych. Dane uzyskane w wyniku badań przedstawione zostały w tabelach numer 8, 9 oraz 10, natomiast w graficzny sposób przedstawione zostały na rysunku 127.

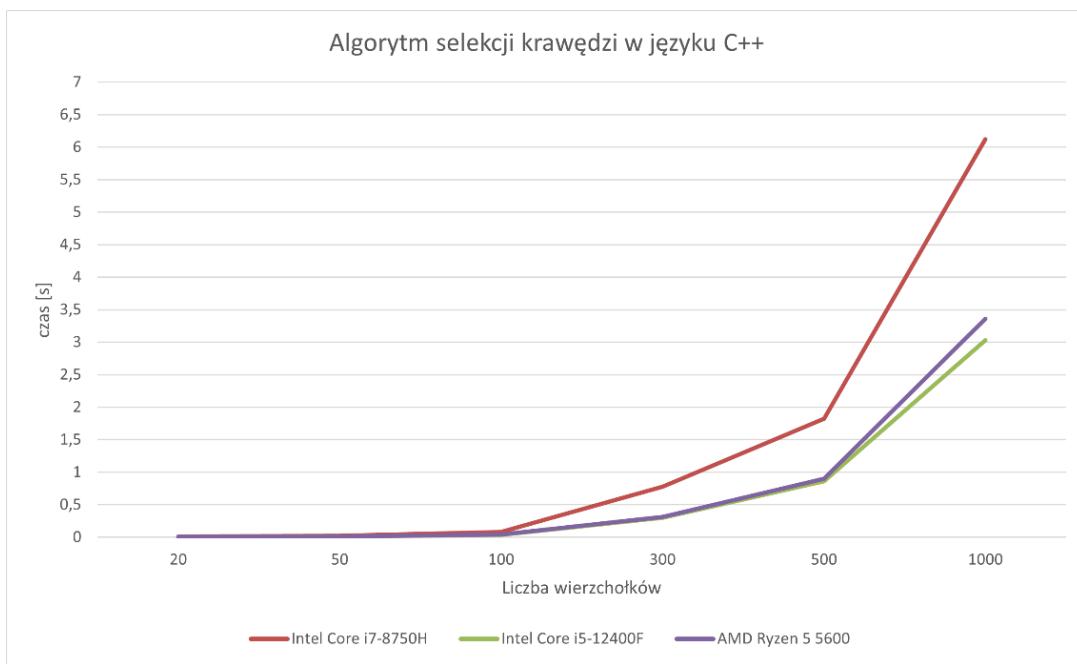


Rysunek 127. Wykres uzyskanych wyników dla algorytmu Christofidesa w języku C++
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 127, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas osiągnęły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Nieznacznie większy wynik uzyskał komputer z procesorem AMD Ryzen 5 5600. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku laptopa posiadającego procesor Intel Core i7-8750H. W celu zbadania złożoności obliczeniowej algorytmu Christofidesa przeprowadzona została analiza teoretyczna, którą rozpoczęło wczytanie danych wejściowych z pliku, a dokładniej ilości wierzchołków N oraz krawędzi M. Złożoność tego procesu wyniosła $O(M)$. Następnie algorytm dokonał inicjalizacji macierzy sąsiedztwa W, której rozmiar wyniósł $N \cdot N$, co przełożyło się na $O(N^2)$ czasu. W kolejnym kroku obliczona została złożoność algorytmu Prima występującego

w zaimplementowanym programie, który odnajduje minimalne drzewo rozpinające. Złożoność tego procesu wyniosła $O(V^2)$ z wykorzystaniem macierzy sąsiedztwa. Wystąpiła przy tym możliwość do dokonania redukcji do $O(E + V \log V)$, za pomocą wykorzystania kolejki priorytetowej. Następnie dokonano oceny złożoności obliczeniowej dla funkcji znajdowania minimalnego doskonałego skojarzenia dla wierzchołków o stopniu nieparzystym. Złożoność tego procesu wyniosła $O(V^3)$, ze względu na zastosowaną metodę zachłanną. Konstrukcja cyklu Eurela w powstały grafie wyniosła $O(V^2)$, z kolei stworzenie z Eurela cyklu Hamiltona, poprzez pominięcie powtarzających się wierzchołków również wyniosło $O(V^2)$. W fazie końcowej algorytm dokonał obliczenia sum wag krawędzi w cyklu Hamiltona, a złożoność tego etapu wyniosła $O(V)$. Najbardziej złożoną operacją w implementacji algorytmu okazało się być odnajdywanie minimalnego doskonałego skojarzenia, co dało ogólną złożoność algorytmu równą $O(V^3)$. Na podstawie danych z eksperymentów przeprowadzonych na różnych procesorach potwierdzono złożoność algorytmu. Dla przykładu wynik dla 1000 wierzchołków uzyskany na procesorze Intel Core i7-8750H wyniósł 0,80 sekundy. Z kolei dla procesora Intel Core i5-12400F było to już 0,47 sekundy, a dla AMD Ryzen 5 5600 0,61 sekundy. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, zaprezentowany na rysunku 127 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(V^3)$.

W kolejnym etapie przeprowadzano ostatnią analizę porównawczą dla algorytmu zaimplementowanego w języku C++, którym był algorytm selekcji krawędzi. Badanie przeprowadzone zostało na trzech stanowiskach komputerowych. Dane uzyskane w wyniku badań przedstawione zostały w tabelach numer 8, 9 oraz 10, natomiast w graficzny sposób przedstawione zostały na rysunku 128.

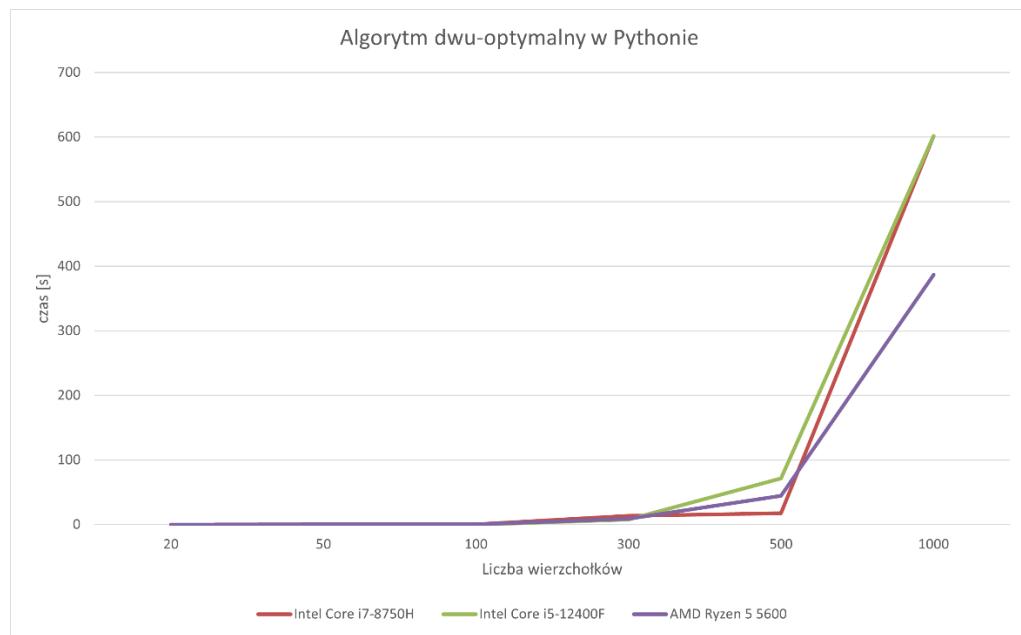


Rysunek 128. Wykres uzyskanych wyników dla algorytmu selekcji krawędzi w języku C++
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 128, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas osiągnęły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Nieznacznie większy wynik uzyskał komputer z procesorem AMD Ryzen 5 5600. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku laptopa posiadającego procesor Intel Core i7-8750H. W celu zbadania złożoności obliczeniowej algorytmu selekcji krawędzi przeprowadzona została analiza teoretyczna, którą rozpoczęło wczytanie danych wejściowych z pliku, a dokładniej ilości wierzchołków N oraz krawędzi M. Złożoność tego procesu wyniosła $O(M)$. Następnie algorytm zainicjalizował struktury danych potrzebne do przechowywania pobranych z pliku danych. Koszt tego procesu wyniosł $O(N)$. W kolejnym etapie program dodał każdą krawędź do multibioru krawędzi wraz z sortowaniem, czego złożoność obliczeniowa wyniosła $O(M \log M)$. Algorytm kolejno dokonał sprawdzenia czy dodanie krawędzi nie spowoduje utworzenia cyklu w grafie. Operacja wykonana została dla każdej krawędzi, a jej zawiłość wyniosła $O(N)$. Kolejny etap polegał na budowie minimalnego drzewa rozpinającego dodając krawędzie, które nie tworzą cykli. Złożoność tego procesu wyniosła $O(M \log M + N^2)$ i stanowi najbardziej złożoną operację w algorytmie selekcji krawędzi. Na podstawie danych z eksperymentów przeprowadzonych na różnych procesorach potwierdzono złożoność algorytmu. Dla przykładu wynik dla 1000 wierzchołków uzyskany na procesorze Intel Core i7-8750H wyniósł 6,12 sekund. Z kolei dla procesora Intel Core i5-12400F było to już 3,03 sekund, a dla AMD Ryzen 5 5600 3,36 sekund.

Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, zaprezentowany na rysunku 128 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(M \log M + N^2)$.

Kolejna część badań objęła przeprowadzenie analizy porównawczej na tych samych algorytmach, zaimplementowanych za pomocą języka Python. Analogicznie przeprowadzanie badań rozpoczęto od algorytmu dwu-optymalnego, na trzech różnych stanowiskach komputerowych. Dane uzyskane w wyniku przeprowadzonych eksperymentów przedstawione zostały w tabelach 11, 12, 13, natomiast w sposób graficzny przedstawione zostały na rysunku 129.

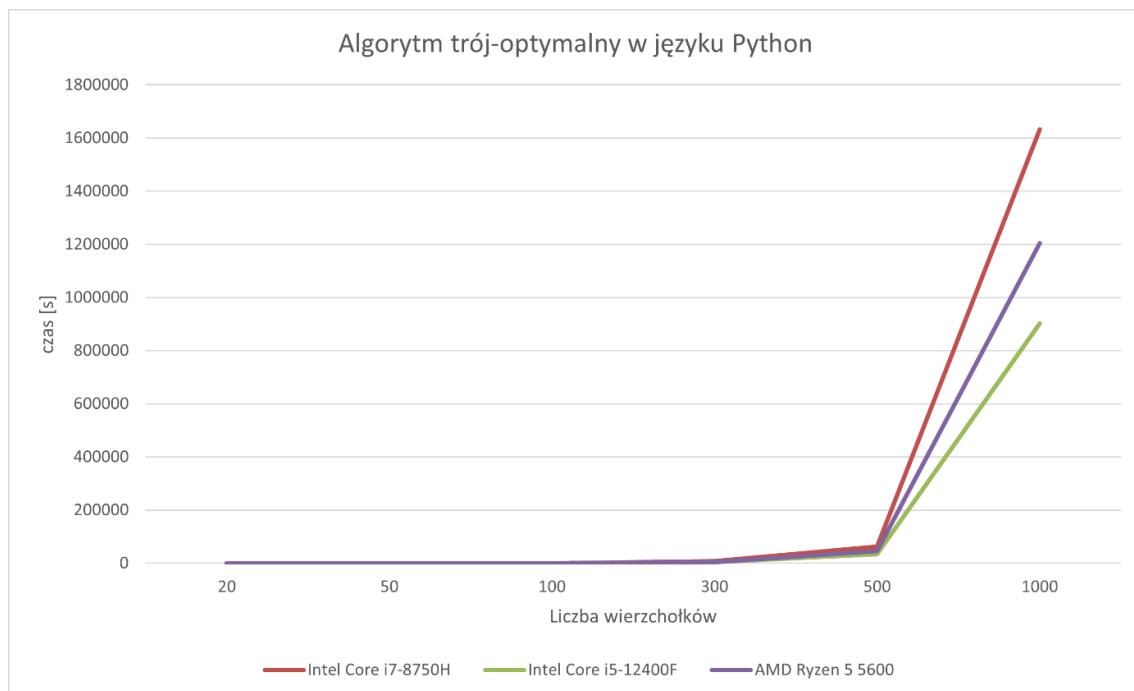


Rysunek 129. Wykres uzyskanych wyników dla algorytmu dwu-optymalnego w Pythonie
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 129, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Procesor Intel Core i5-12400F osiągnął najniższy czas dla 20,50,100 oraz 300 wierzchołków. Laptop posiadający Intel Core i7-8750H dla 500 wierzchołków, z kolei AMD Ryzen 5 5600 dla 1000. Z kolei najdłuższy czas dla wierzchołków 20 oraz 50 osiągnął procesor AMD, dla 100,300 i 1000 laptop z procesorem Intel Core i7-8750H, a dla 500 wierzchołków komputer z procesorem Intel Core i5-12400F. Złożoność obliczeniowa została wykonana analogicznie jak w przypadku implementacji C++ i wyniosła $O(N^4)$. Przykładowo wynik dla 1000 wierzchołków dla Intel Core i7-8750H oraz Intel Core i5-12400F czas wyniósł 601,74 sekund, natomiast dla AMD Ryzen 5 5600

386,40 sekund. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, przedstawiony na rysunku 129 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(N^4)$.

W kolejnym etapie przeprowadzona została analiza porównawcza dla algorytmu trój-optymalnego zaimplementowanego w języku Python. Badanie przeprowadzone zostało na trzech stanowiskach komputerowych. Dane uzyskane w wyniku badań przedstawione zostały w tabelach numer 11, 12 oraz 13, natomiast w sposób graficzny przedstawione zostały na rysunku 130.

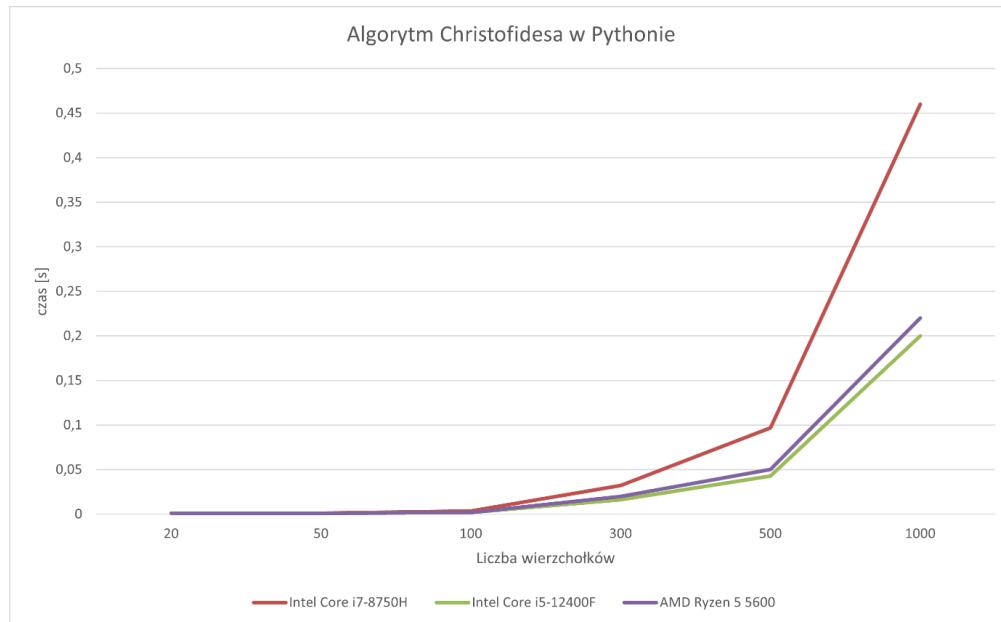


Rysunek 130. Wykres uzyskanych wyników dla algorytmu trój-optymalnego w Pythonie.
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 130, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas osiągnęły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku laptopa posiadającego procesor Intel Core i7-8750H. W celu zbadania złożoności obliczeniowej algorytmu przeprowadzona została analiza teoretyczna, która rozpoczęła się od odczytu danych wejściowych z pliku, łącznie z liczbą wierzchołków N oraz krawędzi M . Złożoność tego procesu wyniosła $O(M)$. Następnie algorytm zainicjował macierz sąsiedztwa W , w której rozmiar wyniósł $N \cdot N$. Proces ten zajął $O(N^2)$ czasu. W kolejnym etapie algorytm zainicjował

tablicę Ptr oraz Route oraz obliczył początkową wagę trasy Tweight . Proces ten miał złożoność $O(N)$. Główna część algorytmu obejmowała iteracyjną optymalizację trasy przy użyciu podejścia 3-opt, które polegało na sprawdzeniu wszystkich możliwych trójek krawędzi w celu znalezienia najlepszej wymiany. Zewnętrzna pętla kontynuowała, dopóki znajdowane były ulepszenia, co mogło trwać do $O(N^3)$ iteracji, ze względu na złożoność sprawdzania wszystkich możliwych trójek krawędzi. W każdej iteracji, sprawdzanie wszystkich trójek krawędzi (dla I , J i K) miało złożoność $O(N^3)$. W najgorszym przypadku, każda iteracja mogła wymagać aktualizacji trasy, co zajmowało $O(N)$ czasu. W rezultacie, łączna złożoność obliczeniowa algorytmu wyniosła $O(N^4)$. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, zilustrowany na rysunku 130 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(N^4)$.

Następnie przeprowadzona została analiza porównawcza dla algorytmu Christofidesa, zaimplementowanego w języku Python. Badanie przeprowadzone zostało na trzech stanowiskach komputerowych. Dane uzyskane w wyniku badań przedstawione zostały w tabelach numer 11, 12 oraz 13, natomiast w sposób graficzny przedstawione zostały na rysunku 131.

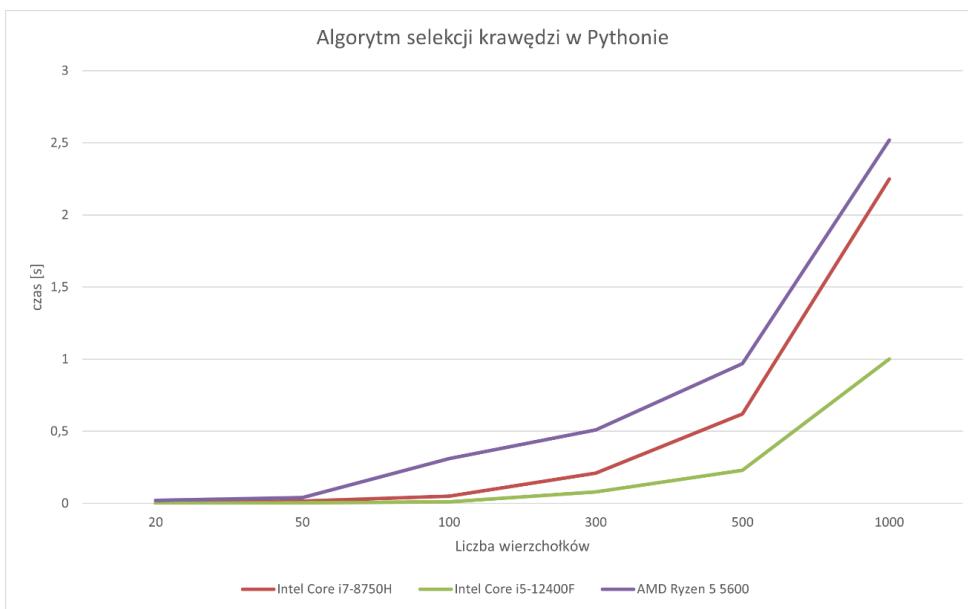


Rysunek 131. Wykres uzyskanych wyników dla algorytmu Christofidesa w Pythonie.
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 131, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas

osiągnęły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Nieznacznie większy wynik uzyskał komputer z procesorem AMD Ryzen 5 5600. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku laptopa posiadającego procesor Intel Core i7-8750H. Złożoność obliczeniowa została analogicznie jak w przypadku implementacji C++. Najbardziej złożoną operacją w implementacji algorytmu okazało się być odnajdywanie minimalnego doskonałego skojarzenia, co dało ogólną złożoność algorytmu równą $O(V^3)$. Na podstawie danych z eksperymentów przeprowadzonych na różnych procesorach potwierdzono złożoność algorytmu. Dla przykładu wynik dla 1000 wierzchołków uzyskany na procesorze Intel Core i7 8750H wyniósł 0,46 sekundy. Z kolei dla procesora Intel Core i5-12400F było to już 0,20 sekundy, a dla AMD Ryzen 5 5600 0,22 sekundy. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, zaprezentowany na rysunku 131 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(V^3)$.

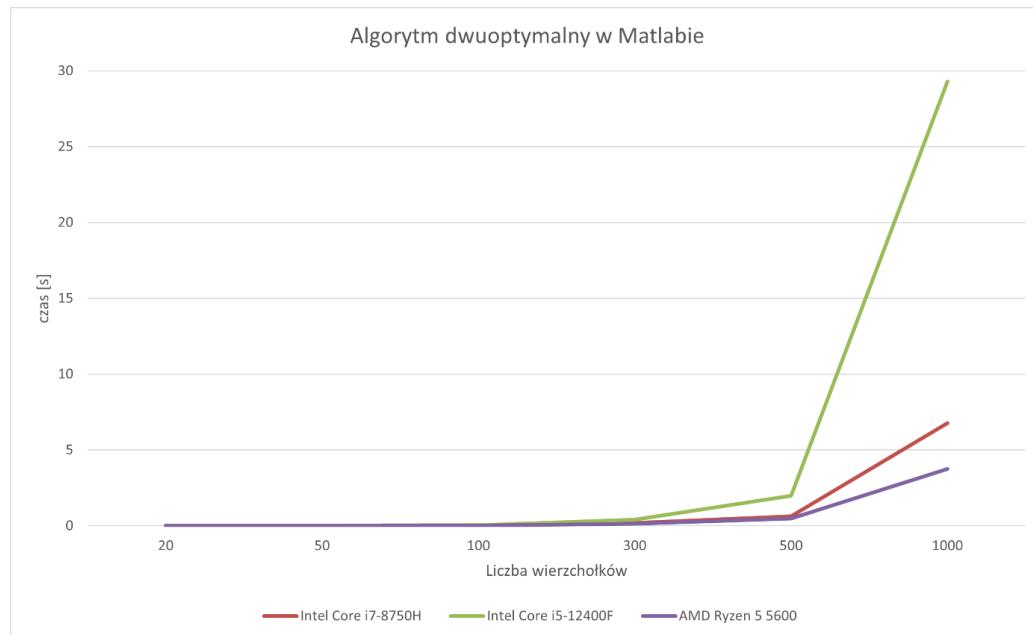
Na koniec przeprowadzania analizy statystycznej na programach opracowanych w języku Python przeprowadzono testy na algorytmie selekcji krawędzi. Badanie przeprowadzone zostało na trzech stanowiskach komputerowych. Dane uzyskane w wyniku badań przedstawione zostały w tabelach numer 11, 12 oraz 13, natomiast w sposób graficzny przedstawione zostały na rysunku 132.



Rysunek 132. Wykres uzyskanych wyników dla algorytmu selekcji krawędzi w Pythonie.
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 132, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas osiągnęły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku komputera wyposażonego w procesor AMD Ryzen 5 5600. Złożoność obliczeniowa została analogicznie jak w przypadku implementacji C++ i wyniosła $O(M \log M + N^2)$. Na podstawie danych z eksperymentów przeprowadzonych na różnych procesorach potwierdzono złożoność algorytmu. Dla przykładu wynik dla 1000 wierzchołków uzyskany na procesorze Intel Core i7-8750H wyniósł 2,25 sekund. Z kolei dla procesora Intel Core i5-12400F była to 1 sekunda, natomiast dla AMD Ryzen 5 5600 2,52 sekund. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, zaprezentowany na rysunku 132 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(M \log M + N^2)$.

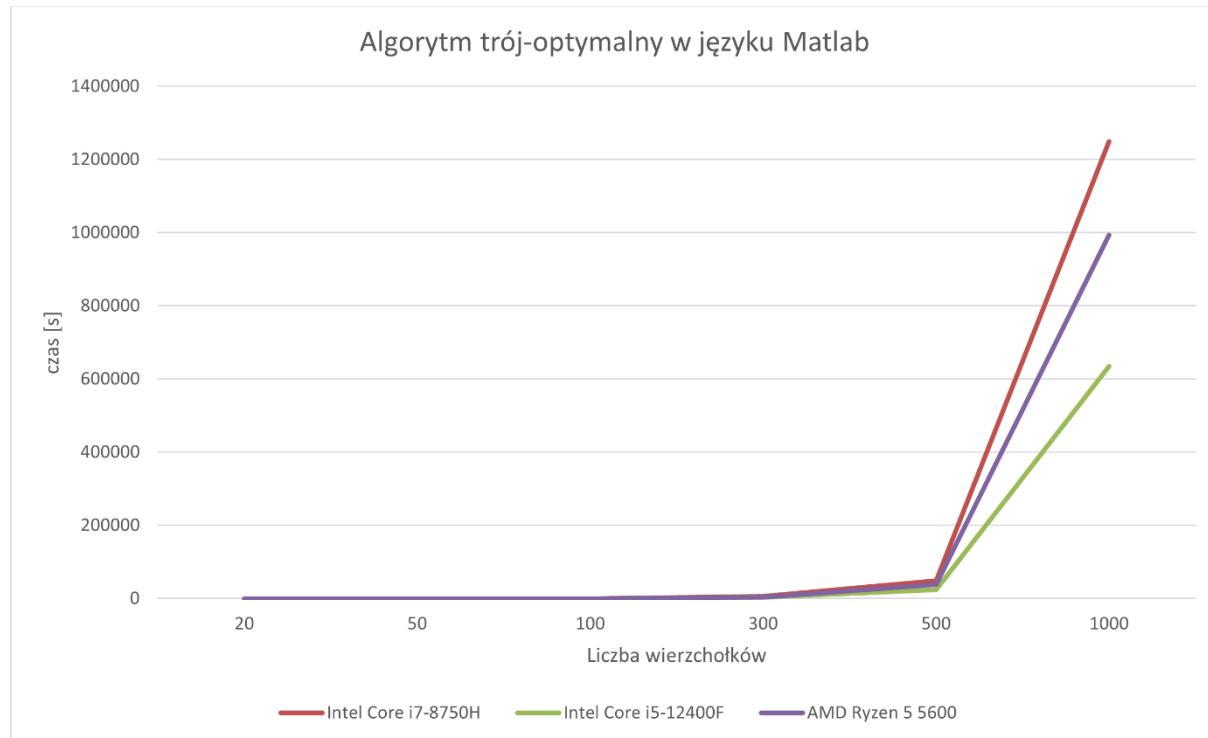
Kolejna część badań objęła przeprowadzenie analizy porównawczej na tych samych algorytmach, zaimplementowanych za pomocą programu Matlab. Analogicznie przeprowadzanie badań rozpoczęto od algorytmu dwu-optymalnego, na trzech różnych stanowiskach komputerowych. Dane uzyskane w wyniku przeprowadzonych eksperymentów przedstawione zostały w tabelach 14, 15, 16, natomiast w sposób graficzny przedstawione zostały na rysunku 133.



Rysunek 133. Wykres uzyskanych wyników dla algorytmu dwu-optymalnego w Matlabie.
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 133, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas osiągnęły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor AMD Ryzen 5 5600. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku komputera wyposażonego w procesor Intel Core i5-12400F. Złożoność obliczeniowa została wykonana analogicznie jak w przypadku implementacji C++ oraz Python i wyniosła $O(N^4)$. Przykładowo wynik dla 1000 wierzchołków dla Intel Core i7-8750H wyniosła 6,77 sekundy. Z kolei Intel Core i5-12400F czas wyniósł 29,28 sekund, natomiast dla AMD Ryzen 5 5600 3,76 sekund. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, przedstawiony na rysunku 133 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(N^4)$.

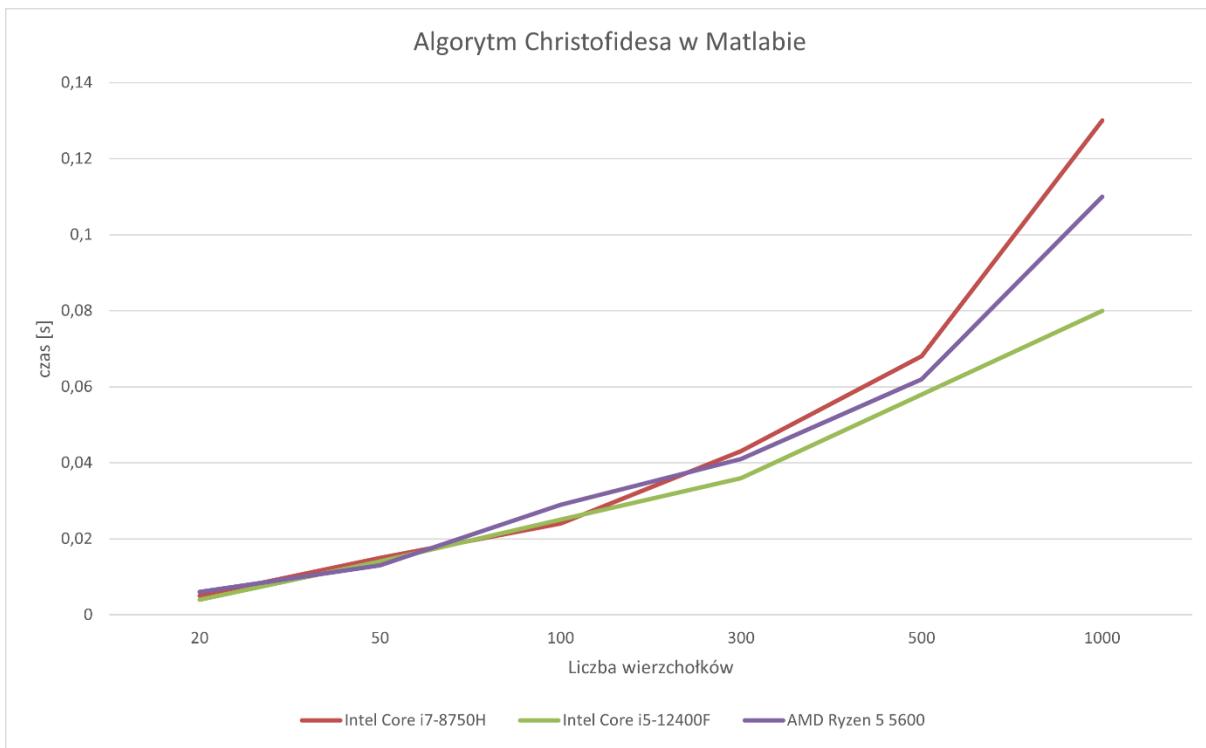
W kolejnym etapie przeprowadzona została analiza porównawcza dla algorytmu trój-optymalnego opracowanego w programie Matlab. Badanie przeprowadzone zostało na trzech stanowiskach komputerowych. Dane uzyskane w wyniku badań przedstawione zostały w tabelach numer 14, 15 oraz 16, natomiast w sposób graficzny przedstawione zostały na rysunku 134.



Rysunek 134. Wykres uzyskanych wyników dla algorytmu trój-optymalnego w Matlabie.
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 134, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas osiągnięły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku laptopa posiadającego procesor Intel Core i7-8750H. W celu zbadania złożoności obliczeniowej algorytmu przeprowadzona została analiza teoretyczna, która rozpoczęła się od odczytu danych wejściowych z pliku, łącznie z liczbą wierzchołków N oraz krawędzi M . Złożoność tego procesu wyniosła $O(M)$. Następnie algorytm zainicjował macierz sąsiedztwa W , w której rozmiar wyniósł $N \cdot N$. Proces ten zajął $O(N^2)$ czasu. W kolejnym etapie algorytm zainicjował tablice Ptr oraz Route oraz obliczył początkową wagę trasy Tweight . Proces ten miał złożoność $O(N)$. Główna część algorytmu obejmowała iteracyjną optymalizację trasy przy użyciu podejścia 3-opt, które polegało na sprawdzeniu wszystkich możliwych trójek krawędzi w celu znalezienia najlepszej wymiany. Zewnętrzna pętla kontynuowała, dopóki znajdowane były ulepszenia, co mogło trwać do $O(N^3)$ iteracji, ze względu na złożoność sprawdzania wszystkich możliwych trójek krawędzi. W każdej iteracji, sprawdzanie wszystkich trójek krawędzi (dla I, J i K) miało złożoność $O(N^3)$. W najgorszym przypadku, każda iteracja mogła wymagać aktualizacji trasy, co zajmowało $O(N)$ czasu. W rezultacie, łączna złożoność obliczeniowa algorytmu wyniosła $O(N^4)$. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, zilustrowany na rysunku 134 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(N^4)$.

Następnie przeprowadzona została analiza porównawcza dla algorytmu Christofidesa, opracowanego w programie Matlab. Badanie przeprowadzone zostało na trzech stanowiskach komputerowych. Dane uzyskane w wyniku badań przedstawione zostały w tabelach numer 14, 15 oraz 16, natomiast w sposób graficzny przedstawione zostały na rysunku 135.

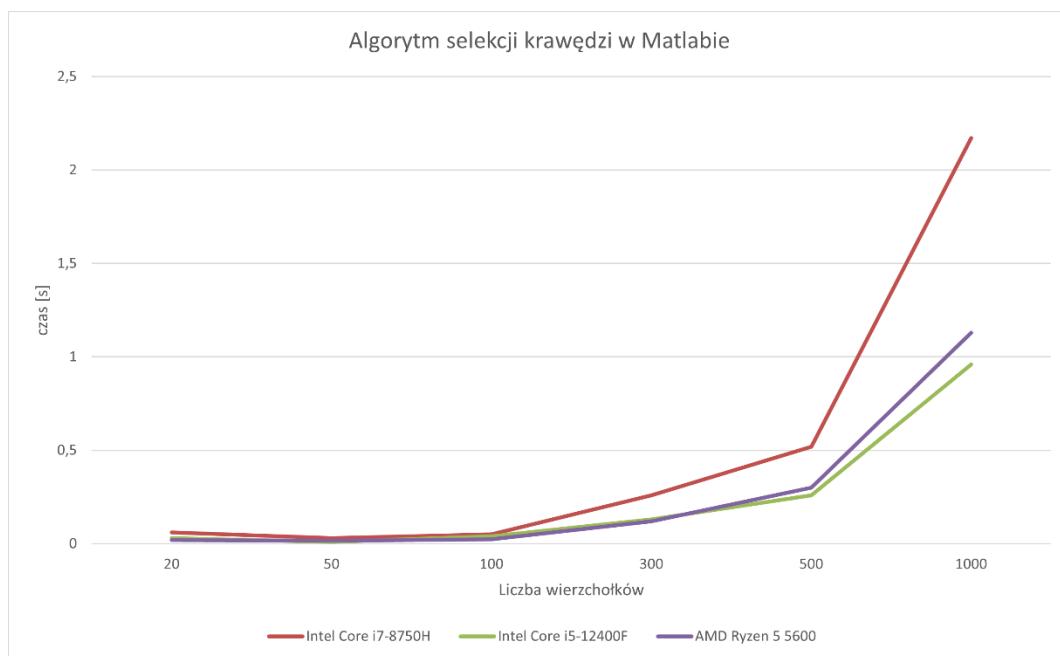


Rysunek 135. Wykres uzyskanych wyników dla algorytmu Christofidesa w Matlabie.
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 135, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas osiągnęły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Nieznacznie większy wynik uzyskał komputer z procesorem AMD Ryzen 5 5600. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku laptopa posiadającego procesor Intel Core i7-8750H. Złożoność obliczeniowa została analogicznie jak w przypadku implementacji C++ oraz Python. Najbardziej złożoną operacją w implementacji algorytmu okazało się być odnajdywanie minimalnego doskonałego skojarzenia, co dało ogólną złożoność algorytmu równą $O(V^3)$. Na podstawie danych z eksperymentów przeprowadzonych na różnych procesorach potwierdzono złożoność algorytmu. Dla przykładu wynik dla 1000 wierzchołków uzyskany na procesorze Intel Core i7-8750H wyniósł 0,13 sekundy. Z kolei dla procesora Intel Core i5-12400F było to 0,08 sekundy, a dla AMD Ryzen 5 5600 0,11 sekundy. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, zaprezentowany na rysunku 135 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(V^3)$.

Na koniec przeprowadzania analizy statystycznej na programach opracowanych w języku Python przeprowadzono testy na algorytmie selekcji krawędzi. Badanie przeprowadzone

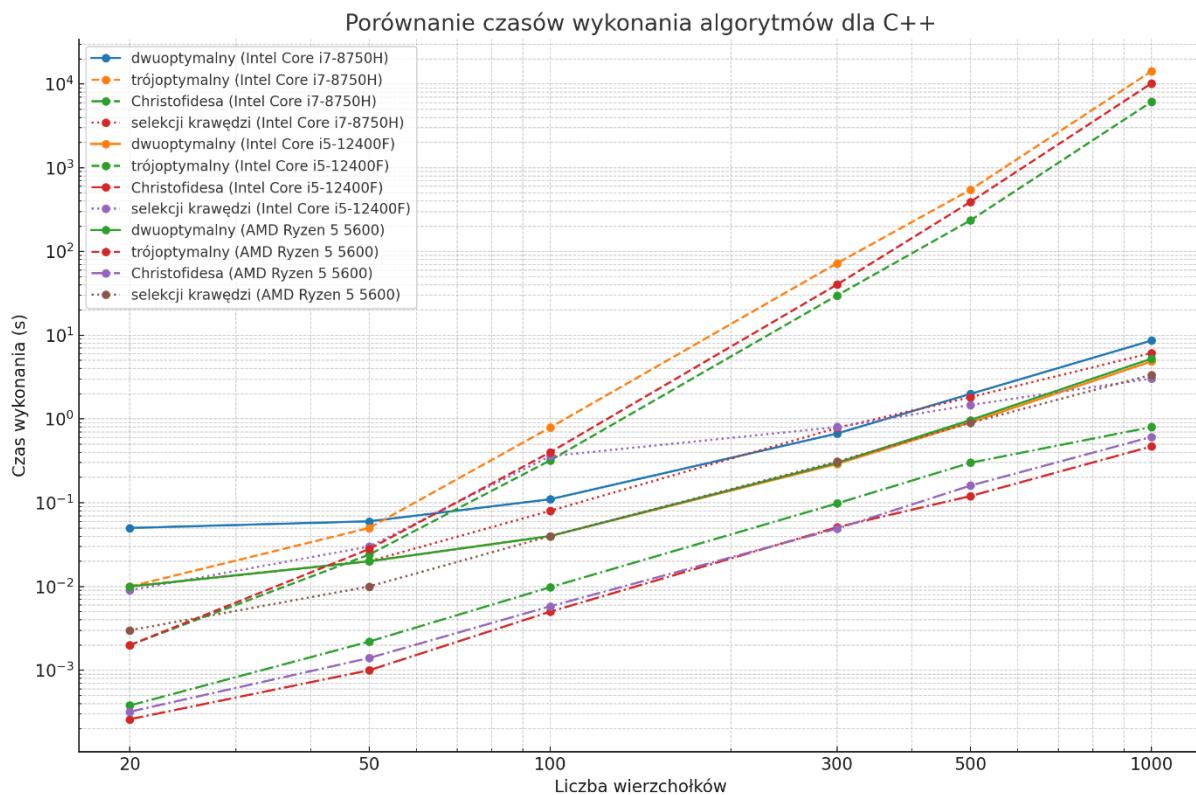
zostało na trzech stanowiskach komputerowych. Dane uzyskane w wyniku badań przedstawione zostały w tabelach numer 14, 15 oraz 16, natomiast w sposób graficzny przedstawione zostały na rysunku 136.



Rysunek 136. Wykres uzyskanych wyników dla algorytmu selekcji krawędzi w Matlabie.
[źródło: opracowanie własne]

Na podstawie wykresu, przedstawionego na rysunku 136, można zaobserwować, że rodzaj parametrów sprzętowych ma znaczący wpływ na działanie programu. Najkrótszy czas osiągnęły programy uruchomione na komputerze stacjonarnym wyposażonym w procesor Intel Core i5-12400F. Najdłuższy czas oczekiwania na wyniki miał miejsce w przypadku komputera wyposażonego w procesor AMD Ryzen 5 5600. Złożoność obliczeniowa została analogicznie jak w przypadku implementacji C++ oraz Python i wyniosła $O(M \log M + N^2)$. Na podstawie danych z eksperymentów przeprowadzonych na różnych procesorach potwierdzono złożoność algorytmu. Dla przykładu wynik dla 1000 wierzchołków uzyskany na procesorze Intel Core i7-8750H wyniósł 2,17 sekund. Z kolei dla procesora Intel Core i5-12400F było to 0,96 sekundy, natomiast dla AMD Ryzen 5 5600 1,13 sekund. Wykres przedstawiający czas wykonania algorytmu w zależności od liczby wierzchołków dla trzech różnych procesorów, zaprezentowany na rysunku 136 pokazuje, że czas wykonania rośnie wykładniczo wraz ze wzrostem liczby wierzchołków, co jest zgodne z teoretycznym oszacowaniem złożoności $O(M \log M + N^2)$.

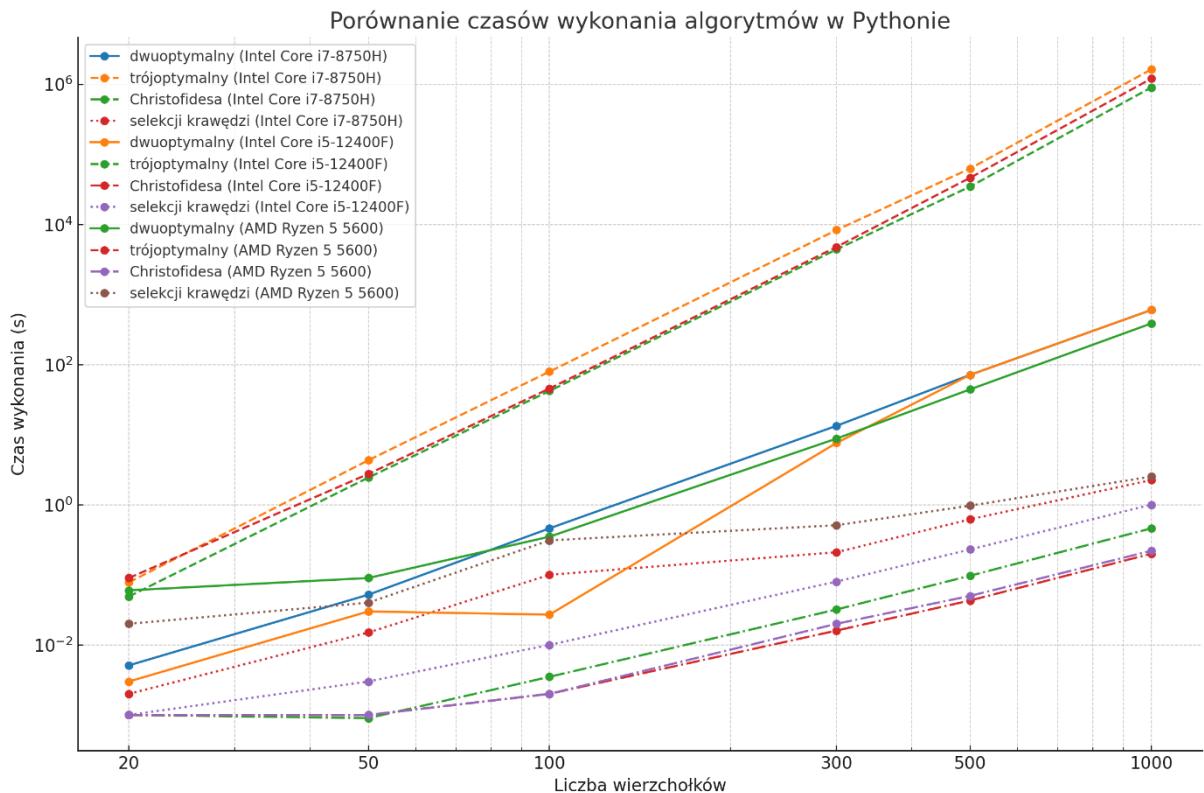
W kolejnym etapie badań przeprowadzone zostało porównanie czasów wykonania algorytmów dla każdego z środowisk programistycznych. Na początku rozpoczęto od sporządzenia wykresu dla języka C++.



Rysunek 137. Wykres porównania czasów wykonania algorytmów w C++.
[źródło: opracowanie własne]

Na wykresie (rysunek 137) przedstawiono porównanie czasów wykonania różnych algorytmów dla trzech procesorów: Intel Core i7-8750H, Intel Core i5-12400F oraz AMD Ryzen 5 5600. Na osi poziomej uwzględniono liczbę wierzchołków (20, 50, 100, 300, 500, 1000), a na osi pionowej czas wykonania w sekundach, w skali logarytmicznej. Porównano cztery algorytmy: dwu-optymalny, trój-optymalny, Christofidesa oraz selekcji krawędzi. Na wykresie zauważono, że czas wykonania algorytmu trój-optymalnego znacznie wzrasta wraz z liczbą wierzchołków, zwłaszcza przy większych wartościach. Algorytm Christofidesa wykazał najkrótszy czas wykonania spośród wszystkich algorytmów dla wszystkich procesorów. Algorytm dwu-optymalny oraz selekcji krawędzi osiągnęły wyniki pośrednie. Różnice w czasach wykonania algorytmów były wyraźnie widoczne między poszczególnymi procesorami, przy czym Intel Core i5-12400F generalnie osiągnął lepsze czasy wykonania w porównaniu do Intel Core i7-8750H oraz AMD Ryzen 5 5600.

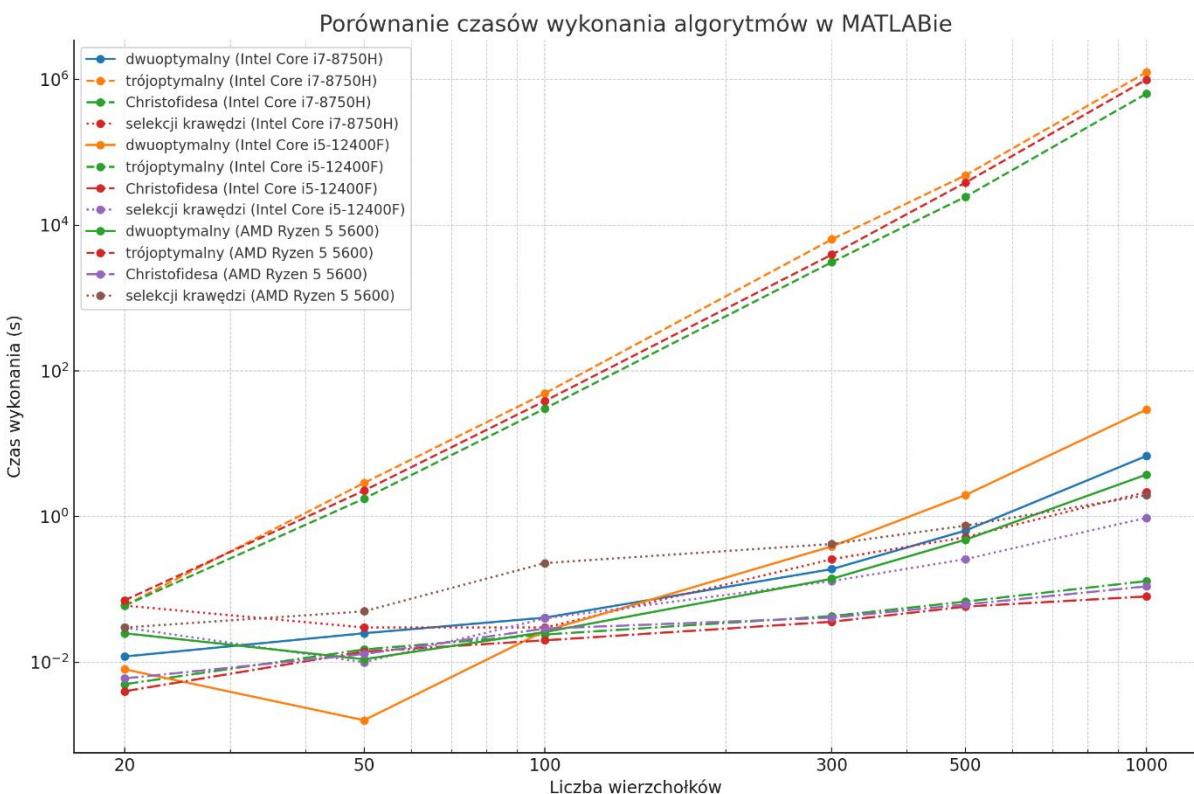
W sposób analogiczny przeprowadzone zostało porównanie czasów wykonania algorytmów dla języka Python.



Rysunek 138. Wykres porównania czasów wykonania algorytmów w Pythonie.
[źródło: opracowanie własne]

Rysunek 138 przedstawia porównanie czasów wykonania różnych algorytmów dla trzech procesorów. W wyniku przeprowadzonego badania zauważono, że czas wykonania algorytmu trój-optymalnego znaczco wzrasta wraz z liczbą wierzchołków, szczególnie przy większych wartościach. Algorytm Christofidesa wykazał najkrótszy czas wykonania spośród wszystkich algorytmów dla wszystkich procesorów. Algorytm dwu-optymalny oraz selekcji krawędzi osiągnęły wyniki pośrednie.

Porównanie czasów wykonania algorytmów wykonane zostało również dla programu Matlab, co zostało przedstawione na rysunku 139.



Rysunek 139. Wykres porównania czasów wykonania algorytmów w Matlabie.
[źródło: opracowanie własne]

Na powyższym wykresie przedstawiono porównanie czasów wykonania różnych algorytmów w Matlabie dla trzech procesorów: Intel Core i7-8750H, Intel Core i5-12400F oraz AMD Ryzen 5 5600. Porównane zostały cztery algorytmy: dwu-optymalny, trój-optymalny, Christofidesa oraz selekcji krawędzi. Na wykresie zauważono, że czas wykonania algorytmu trój-optymalnego znacząco wzrasta wraz z liczbą wierzchołków, szczególnie przy większych wartościach. Algorytm Christofidesa wykazał najkrótszy czas wykonania spośród wszystkich algorytmów dla wszystkich procesorów. Algorytm dwuoptymalny oraz selekcji krawędzi osiągnęły wyniki pośrednie.

W ramach realizacji celu zakresu pracy dokonano oceny złożoności obliczeniowej algorytmów. Opracowano wykres średniego czasu wykonania różnych algorytmów, w zależności od używanego procesora oraz ich złożoność obliczeniową. Porównano cztery algorytmy: dwuoptymalny, trójoptymalny, Christofidesa oraz selekcji krawędzi, przy czym każdy z nich posiadał inną złożoność obliczeniową. Na początek zebrano wszystkie zawiłości obliczeniowe w tabeli numer 17.

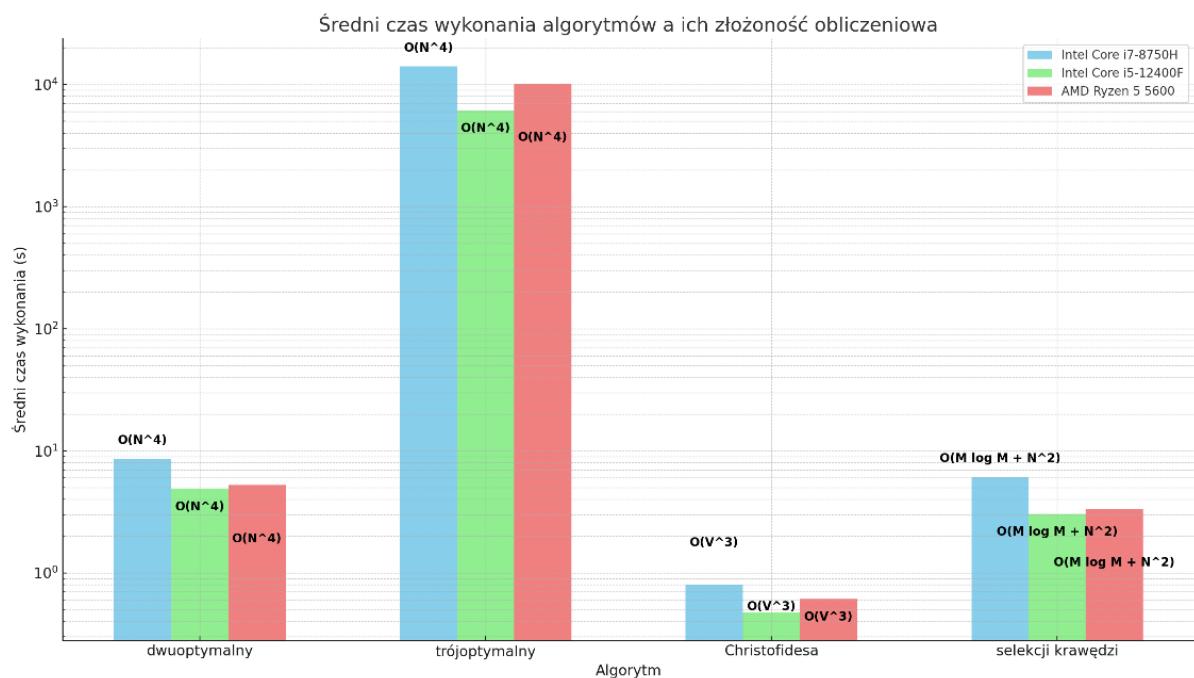
Tabela 17.

Oceny złożoności obliczeniowej każdego algorytmu.

Algorytm	Złożoność obliczeniowa
Dwu-optymalny	$O(N^4)$
Trój-optymalny	$O(N^4)$
Christofidesa	$O(V^3)$
Selekcji krawędzi	$O(M \log M + N^2)$

źródło: opracowanie własne

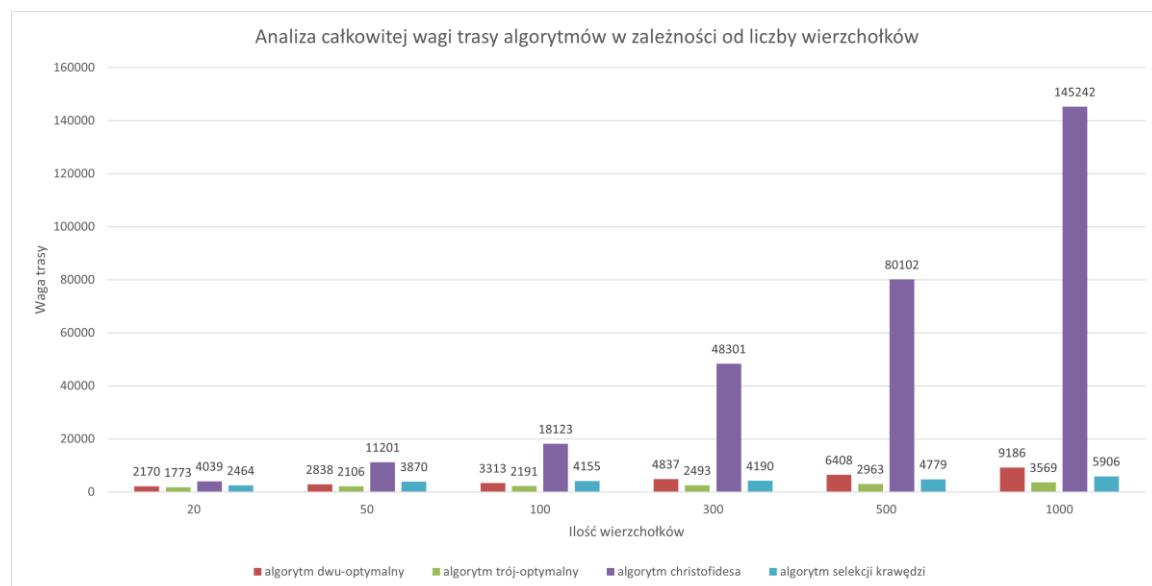
Na podstawie danych dotyczących złożoności przedstawionych w tabeli 17 oraz czasów każdego z algorytmów z tabel numer 8, 9, 10, 11, 12, 13, 14, 15, 16 obliczony został średni czas wykonania różnych algorytmów w zależności od używanego procesora oraz ich złożoność obliczeniową. Średnie czasy wykonania zostały zmierzone dla trzech różnych procesorów: Intel Core i7-8750H, Intel Core i5-12400F oraz AMD Ryzen 5 5600.

Rysunek 140. Wykres złożoności obliczeniowej algorytmów
[źródło: opracowanie własne]

Na wykresie przedstawionym na rysunku 140 zaobserwowano, że algorytm trój-optymalny charakteryzuje się najdłuższym czasem wykonania na wszystkich trzech procesorach, co było zgodne z jego wyższą złożonością obliczeniową $O(N^4)$. Z kolei algorytm Christofidesa, posiadający złożoność $O(V^3)$, wykazuje najkrótszy czas wykonania. Różnice w wysokości słupków dla poszczególnych procesorów wynikają z ich różnych architektur, mocy obliczeniowej oraz efektywności w wykonywaniu danych algorytmów. Procesory różnią się liczbą rdzeni i wątków, taktowaniem, wielkością pamięci cache oraz efektywnością

energetyczną. Dla przykładu algorytm dwu-optymalny zaimplementowany w języku C++ na procesorze Intel Core i5-12400F osiągnął najkrótszy czas wykonania (4.86 sekund), natomiast Intel Core i7-8750H najdłuższy (8.62 sekund). W przypadku algorytmu trój-optymalnego wszystkie procesory miały długi czas wykonania, z Intel Core i7-8750H na czele (14135 sekund). Z kolei w przypadku algorytmu Christofidesa wszystkie posiadają bardzo krótki czas wykonania. W algorytmie selekcji krawędzi Intel Core i5-12400F ponownie posiadał najkrótszy czas (3.03 sekund), a Intel Core i7-8750H najdłuższy (6.12 sekund).

W ramach przeprowadzonych badań dokonana została również analiza całkowitej wagi trasy algorytmów w zależności od liczby wierzchołków, w ramach której zgromadzono dane dotyczące całkowitych wag tras dla każdego z zaimplementowanych algorytmów. Efekt doświadczenia przedstawiony został na rysunku 141.



Rysunek 141. Analiza całkowitej wagi tras algorytmów w zależności od liczby wierzchołków
[źródło: opracowanie własne]

W wyniku przeprowadzonego badania zaobserwowano, że dla mniejszych grafów (20 wierzchołków) algorytmy osiągały stosunkowo niską wagę tras. W miarę wzrostu liczby wierzchołków do 1000, waga tras rośnie dla wszystkich algorytmów. Algorytm Christofidesa, mimo, że był wydajny przy mniejszych grafach, dla większych wykazywał znacznie wyższą wagę tras, co sugerowało, że może być mniej efektywny w tych przypadkach. Algorytm trój-optymalny oraz algorytm selekcji krawędzi charakteryzował się bardziej zrównoważonymi wynikami, utrzymując względnie stabilne wartości wag tras dla różnych rozmiarów grafów. Na podstawie wykresu przedstawionego na rysunku 141, można

wnioskować, który algorytm jest najbardziej efektywny w zależności od rozmiaru problemu, co pozwala na lepsze dopasowanie algorytmu do specyficznych potrzeb.

Przeprowadzając analizę dokonano również wyboru optymalnego algorytmu pod względem czasu oraz wagi trasy. Po przeanalizowaniu wszystkich danych okazało się, że algorytm selekcji krawędzi zachowuje balans pomiędzy czasem, a całkowitą wagą tras. W przypadku badań nad algorytmem Christofidesa, pomiar czasu był problematyczny ze względu na jego szybkie działanie. Konieczne było ustalenie dużej liczby miejsc po przecinku, aby uzyskać dokładny wynik pomiaru czasu. Konieczne było ustalenie dużej liczby miejsc po przecinku, aby uzyskać dokładny wynik pomiaru czasu. Z kolei pod względem wagi trasy stwierdzono, że algorytm trój-optymalny oraz selekcji krawędzi charakteryzuje się bardziej zróżnicowanymi wynikami, utrzymując względnie stabilne wartości wag tras dla różnych rozmiarów grafów.

Znaczący wpływ na uzyskane wyniki miały również stanowiska komputerowe, na których badania zostały przeprowadzone. Laptop wyposażony w procesor Intel Core i7-8750H cechował się stabilną wydajnością wielowątkową oraz dobrą optymalizacją w Windows 11 Home. Uzyskał on lepsze wyniki w zadania wymagających wysokiej przepustowości pamięci. Komputer z procesorem Intel Core i5-12400F charakteryzował się wyższą częstotliwością taktowania oraz większą pamięcią podręczną i pamięcią RAM, co umożliwiło szybsze przetwarzanie danych, co w rezultacie przekładało się na krótsze czasy wykonania algorytmów. Natomiast komputer wyposażony w procesor AMD Ryzen 5 5600 posiadał najwyższą bazową częstotliwość taktowania oraz największą pamięć podręczną zapewniając tym samym szybkie przetwarzanie danych, co okazało się korzystne dla zadań jednowątkowych oraz intensywnych obliczeniowo. Dzięki różnym konfiguracjom sprzętowym można było zaobserwować, jak różne parametry wpływają na efektywność wykonywania algorytmów, co jest kluczowe przy wyborze odpowiedniego sprzętu dla konkretnych zastosowań algorytmicznych.

Wybór odpowiedniego języka programowania ma również bardzo znaczący wpływ na wydajność oraz efektywność realizacji algorytmów. Język C++ stanowił najlepszy wybór dla zadań wymagających najwyższej wydajności i precyzyjnej kontroli nad zasobami sprzętowymi. Dodatkowo warto zaznaczyć, że język ten poradził sobie najlepiej z bardziej złożonymi danymi. W przypadku algorytmu trój-optymalnego, był jedynym językiem, w którym nie wystąpiła potrzeba obliczania szacowanego czasu za pomocą proporcji. Język Python z kolej jest najlepszym wyborem dla szybkiego prototypowania, elastyczności i łatwości implementacji. Matlab natomiast był optymalny dla obliczeń numerycznych i analizy danych, z zaawansowanymi narzędziami do obliczeń macierzowych i wizualizacji.

W przypadku Pythona i Matlaba istniała możliwość uzyskania wyników czasowych dla algorytmu trój-optymalnego, jednak wymagało to łącznie ponad 1900 godzin ciągłej pracy wszystkich jednostek.

Przeprowadzone badania, poprzedzone przeglądem wszelkiej literatury związanego z tematem komiwojażera dowiodły, że na wyniki wpływ mają różne czynniki m.in. środowisko programistyczne, ilość oraz rodzaj danych, złożoność obliczeniowa implementacji oraz parametry jednostek, na których przeprowadzane zostały testy. Przegląd literatury wykazał, że brakuje informacji na temat rozwiązymania problemu komiwojażera za pomocą metody selekcji krawędzi. Dodatkowo podjęto próbę implementacji tego algorytmu za pomocą języka programowania Python. W ramach eksperymentu oraz znacznych braków w próbach rozwiązyania wyselekcjonowanych algorytmów za pomocą środowiska obliczeń naukowych w literaturze, postanowiono wykonać implementacje oraz badania również za pomocą programu Matlab. Wyniki badań pozwoliły ustalić, że najszybszy podczas przeprowadzania testów był algorytm Christofidesa, z kolei uwzględniając rezultaty całkowitej wagi trasy optymalnym wyborem był algorytm trój-optymalny oraz selekcji krawędzi. Analiza wszystkich danych pod względem zachowania równowagi pomiędzy czasem a całkowitą wagą tras wykazała, że optymalnym wyborem jest algorytm selekcji krawędzi. Dowiedziono, że optymalnym językiem do przeprowadzania tego typu testów jest język programowania C++, ponieważ poradził sobie najlepiej z bardziej złożonymi obliczeniami. W przypadku bardziej złożonego algorytmu, jakim był algorytm trój-optymalny, C++ w najkrótszym czasie uzyskał wyniki dla większej ilości wierzchołków, bez konieczności wyliczania szacowanego czasu za pomocą proporcji, co miało miejsce w przypadku środowiska Pythona oraz Matlaba. Reasumując kwintesencję tematu, warto zastanowić się, jakie wyniki uzyskane zostałyby w przypadku zbadania większej ilości danych, przykładowo dla 5000, czy 10000 wierzchołków. Tego typu badanie nie zostało przeprowadzone w wyniku analizy porównawczej niniejszej pracy, ze względu na ograniczone zasoby czasowe oraz sprzętowe. Z pewnością przełożyłoby się to na dłuższy czas ciągłej pracy każdej z jednostek. Można uznać, że osoby chętne odtworzenia we własnym zakresie wszystkich omówionych badań uzyskają zbliżone, takie same, bądź nawet lepsze wyniki, jednak znaczący wpływ na to miałaby ilość oraz rodzaj danych, środowisko programistyczne oraz specyfikacja sprzętowe jednostki komputerowej. Warto podjąć dyskusję na ten temat.

9. Podsumowanie i wnioski

Celem pracy badawczej było przeprowadzenie analizy algorytmów rozwiązujących problem komiwojażera. W wyniku przedstawienia teoretycznego wybranych algorytmów, dokonano selekcji oraz implementacji algorytmów dwu-optymalnego, trój-optymalnego, Christofidesa oraz selekcji krawędzi. Podjęto decyzję o opracowaniu algorytmów w różnych środowiskach programistycznych tj. za pomocą języka programowania C++ i Python oraz programu Matlab, aby zweryfikować jaki wpływ na działanie algorytmu ma dobór środowiska implementacji. Dodatkowo zweryfikowany został wpływ parametrów sprzętowych trzech różnych stanowisk komputerowych na efektywność działania algorytmów. W ramach realizacji celów pracy dyplomowej opracowany został również generator grafów losowych, za pomocą którego opracowane zostały dane wejściowe do przeprowadzenia badań. Wygenerowane zostały dane dla 20, 50, 100, 300, 500 oraz 1000 wierzchołków, w celu przeprowadzenia analizy każdego z algorytmów na tych samych danych wejściowych. Wyniki uzyskane podczas przeprowadzania testów oraz kod źródłowy każdego z algorytmów, umożliwiły ocenę złożoności obliczeniową każdego z nich. Pod względem zawiłości obliczeniowej najlepszym wyborem okazał się algorytm trój-optymalny, którego złożoność wyniosła $O(N^4)$. Najszybszym algorytmem we wszystkich środowiskach programistycznych był algorytm Christofidesa. Jednak ze względu na jego szybkość, pomiar czasu działania okazał się problematyczny i wymagał ustawienia dużej liczby miejsc po przecinku. Z kolei, biorąc pod uwagę wyłącznie wagi tras, stwierdzono, że algorytmy trój-optymalny oraz selekcji krawędzi charakteryzują się bardziej zróżnicowanymi wynikami, jednocześnie utrzymując względnie stabilne wartości wag tras dla różnych rozmiarów grafów. Optymalnym algorytmem zachowującym bilans pomiędzy czasem, a całkowitą wagą tras okazał się być algorytm selekcji krawędzi.

Reasumując kwintesencję tematu, doszło do fundamentalnej konkluzji i dowiedziono, że wybór najlepszego algorytmu do rozwiązania problemu komiwojażera jest zależny od wymagań dotyczących złożoności obliczeniowej, doboru środowiska programistycznego oraz parametrów sprzętowych, na których algorytmy będą testowane. Warto zaznaczyć, że wpływ na wyniki miał również rodzaj oraz ilość danych wejściowych. Wyniki przeprowadzonych badań dostarczyły istotnych informacji, które mogłyby zostać wykorzystane zarówno w teoretycznym, jak i praktycznym podejściu do NP-trudnego problemu komiwojażera. Dane te mogły posłużyć jako baza do rozwijania nowych algorytmów, weryfikacji istniejących metod optymalizacji oraz do oceny efektywności różnych heurystyk w kontekście rozwiązywania problemu komiwojażera.

Bibliografia

Wykaz literatury:

- Anholcer, M. (2023). *Badania operacyjne*. Poznań: Uniwersystet Ekonomiczny w Poznaniu.
- Bell, P. i Brent, B. (2015). *GitHub. Przyjazny przewodnik*. Gliwice: Helion.
- Cook, W. J. (2012). *In Pursuit of the Traveling Salesman : Mathematics at the Limits of Computation*. Princeton: Princeton University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L. i Stein, C. (2007). *Wprowadzenie do algorytmów*. Warszawa: WTN.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L. i Stein, C. (2009). *Introduction to Algorithms*. London: Massachusetts Institute of Technology.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L. i Stein, C. (2018). *Wprowadzenie do algorytmów*. Warszawa: PWN.
- Debudaj-Grabysz, A., Deorowicz, S. i Widuch, J. (2012). *Algorytmy i struktury danych - Wybór zaawansowanych metod*. Gliwice: Wydawnictwo Politechniki Śląskiej.
- Dorigo, M. i Stützle, T. (2004). *Ant Colony Optimization*. London: MIT Press.
- Gilat, A. (2016). *MATLAB: An Introduction with Applications - Sixth Edition*. Hoboken, New Jersey: Willey.
- Ladner, R. E. (1975). *On the structure of polynomial time reducibility*. Journal of the ACM.
- Martelli, A., Martelli Ravenscroft, A., Holden, S. i McGuire, P. (2023). *Python w pigulce - Podręczny przewodnik po wersjach 3.10 i 3.11*. Warszawa: APN Promise.
- Strąk, Ł. L. (2017, Luty 06). Adaptacyjny algorytm optymalizacji stadnej cząsteczek dla dynamicznego problemu komiwojażera. Praca doktorska. Katowice : Uniwersytet Śląski. Katowice, Śląskie, Polska.
- Sysło, M., Deo, N. i Kowalik, J. (1995). *Algorytmy optymalizacji dyskretnej z programami w języku Pascal*. Warszawa: PWN.
- VanderPlas, J. (2023). *Python Data Science. Niezbędne narzędzia do pracy z danymi. Wydanie II*. Gliwice: Helion.
- Wilson, R. J. (2000). *Wprowadzenie do teorii grafów*. Warszawa: PWN.

- Wojciechowski, J. i Pieńkocz, K. (2013). *Grafy i sieci*. Warszawa: PWN.
- Zieliński, J. (2013). *Podstawy programowania w języku C++*. Kraków: Impuls.
- Zieliński, J. (2024). Heurystyczny algorytm komiwojażera oparty na selekcji krawędzi. (*Artykuł niepublikowany*).
- Źródła internetowe:
- abcdef.wiki. (2024, Maj 20). https://pl.abcdef.wiki/wiki/Metric_%28mathematics%29
- Błaszkiewicz, D. (2011, Maj 11). *Instytut Informatyki Uniwersytetu Wrocławskiego*. https://ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/alg_mrow.opr.pdf
- Brilliant.org. (2024, Maj 08). <https://brilliant.org/wiki/complexity-classes/>
- CodingDrills. (2024, Maj 15). <https://www.codingdrills.com/tutorial/introduction-to-greedy-algorithms/traveling-salesman>
- EconPapers. (2023, Listopad 11). https://econpapers.repec.org/article/inmoropre/v_3a11_3ay_3a1963_3ai_3a6_3ap_3a972-989.htm
- Encyklopedia Algorytmów. (2017, Czerwiec 17). http://algorytmy.ency.pl/artykul/k_opt
- Encyklopedia Algorytmów. (2020, Maj 1). http://algorytmy.ency.pl/artykul/algorytmy_heurystyczne
- GitHub Docs. (2024, Czerwiec 13). <https://docs.github.com/en/desktop/overview/getting-started-with-github-desktop>
- Kohlstedt, K. (2022, Październik 4). *99percentinvisible*. <https://99percentinvisible.org/article/the-seven-bridge-problem-how-an-urban-puzzle-inspired-a-new-field-of-mathematics/>
- Kowalik. (2011, Czerwiec 29). *Informatyka MIMUW*. <http://smurf.mimuw.edu.pl/node/1122>
- Learn Microsoft. (2024, Maj 21). <https://learn.microsoft.com/en-us/visualstudio/ide/what's-new-visual-studio-2022?view=vs-2022>
- Sidford, A. (2020, Wrzesień 15). *Stanford, University*. https://web.stanford.edu/~sidford/courses/20fa_opt_theory/sidford_2020fa_mse213_cs269o_lec1.pdf#:~:text=URL%3A%20https%3A%2F%2Fweb.stanford.edu%2F~sidfo

rd%2Fcourses%2F20fa_opt_theory%2Fsidford_2020fa_mse213_cs269o_lec1.pdf%0
AVisible%3A%200%25%20

Toksari, M. D. (2015, Grudzień 03). *ScieneDirect*.

<https://www.sciencedirect.com/science/article/abs/pii/S0142061515005840>

Traveling Salesman Problem. (2015, Marzec).

<https://math.uwaterloo.ca/tsp/concorde/index.html>

Vasava, M. (2019, Sierpień 03). *Medium.com*. <https://medium.com/@mitalvasava411/what-is-a-minimum-spanning-tree-2be2b3cec66a>

Weisstein, E. W. (2024, Maj 03). MathWorld - A Wolfram Web Resource:

<https://mathworld.wolfram.com/EulerianCycle.html>

Wikipedia. (2023, Maj 16). https://pl.wikipedia.org/wiki/Zagadnienie_mostów_królewieckich

Wikipedia. (2023, Luty 10). https://pl.wikipedia.org/wiki/Algorytm_aproksymacyjny

Wikipedia. (2023, Grudzień 23). https://en.wikipedia.org/wiki/Concorde_TSP_Solver

Wikipedia. (2023, Maj 28). https://uk.wikipedia.org/wiki/Мурашиний_алгоритм

Wikipedia. (2024, Marzec 25).

https://pl.wikipedia.org/wiki/Twierdzenie_o_czterech_barwach

Wikipedia. (2024, Maj 01). [https://pl.wikipedia.org/wiki/Graf_\(matematyka\)](https://pl.wikipedia.org/wiki/Graf_(matematyka))

Wikipedia. (2024, Marzec 25). https://pl.wikipedia.org/wiki/Twierdzenie_Diraca

Wikipedia. (2024, Marca 01). https://en.wikipedia.org/wiki/Vertex_cover

Wikipedia. (2024, Maj 07). https://en.wikipedia.org/wiki/Computational_complexity_theory

Wikipedia. (2024, Maj 08). https://en.wikipedia.org/wiki/Travelling_salesman_problem

Wikipedia. (2024, Maj 11). https://en.wikipedia.org/wiki/Metric_space

Wikipedia. (2024, Styczeń 08). https://pl.wikipedia.org/wiki/Przestrzeń_euklidesowa

Wikipedia. (2024, Luty 28). https://en.wikipedia.org/wiki/Taxicab_geometry

Wikipedia. (2024, Czerwiec 14). <https://pl.wikipedia.org/wiki/C%2B%2B>

Wikipedia. (2024, Maj 15). <https://pl.wikipedia.org/wiki/Python>

World Traveling Salesman Problem. (2021, Luty 15).

<https://www.math.uwaterloo.ca/tsp/world/>

Yadav, A. (2022, Kwiecień 03). *Medium.com*. <https://aditya-yadav.medium.com/kruskals-algorithm-for-minimum-spanning-tree-f3c0c0b7b386>

Spis rysunków

Rysunek 1. Plan miasta Królewiec z czasów Eulera.	6
Rysunek 2. Przedstawienie mostów w Królewcu według Eulera.	7
Rysunek 3. Wykres przedstawiający reprezentacje Mostów Królewieckich.	8
Rysunek 4. Rozwiązywanie Eurela problemu wędrówki rycerza.	9
Rysunek 5. Trasa rycerska na wykresie szachownicy.	9
Rysunek 6. Icosian.	10
Rysunek 7. Cykl Hamiltona w grafie.	12
Rysunek 8. Pokolorowana mapa cyklu Hamiltona.	12
Rysunek 9. Zoptymalizowana trasa podróży po stanie Maine z 1925 roku.	15
Rysunek 10. Zrzut ekranu z programu Concorde.	17
Rysunek 11. Rozwiązywanie TSP dla 85 900 miast w zastosowaniu chipa komputerowego.	18
Rysunek 12. Przedstawienie małego obszaru trasy obejmującej 85 900 iast.	19
Rysunek 13. Rozwój algorytmów na przestrzeni lat.	19
Rysunek 14. Porównanie metryki Manhattan oraz metryki euklidesowej.	24
Rysunek 15. Hierarchia problemów decyzyjnych.	26
Rysunek 16. Graf eurelowski.	29
Rysunek 17. Graf półeurelowski.	29
Rysunek 18. Graf hamiltonowski.	30
Rysunek 19. Graf półhamiltonowski.	32
Rysunek 20. Graf niehamiltonowski.	32
Rysunek 21. Wymiana 2-optymalna.	40
Rysunek 22. Wymiana trój-optymalna.	41
Rysunek 23. Macierz odległości pomiędzy miastami.	42
Rysunek 24. Proces działania algorytmu Kruskala.	45
Rysunek 25. Proces działania algorytmu Prima.	47
Rysunek 26. Przykładowy graf regularny.	48
Rysunek 27. Proces działania algorytmu selekcji krawędzi.	49
Rysunek 28. Graficzne przedstawienie algorytmu mrówkowego.	57
Rysunek 29. Działanie algorytmu Christofidesa.	61
Rysunek 30. Blokada trasy $\langle W_3, W_1 \rangle$ w celu niedopuszczenia do powstania cyklu.	65
Rysunek 31. Drzewo rozwiązań.	66
Rysunek 32. Ostateczne zamknięte drzewo rozwiązań.	66
Rysunek 33. Interfejs programu Visual Studio Community 2022.	67

Rysunek 34. Przedstawienie interfejsu Jupyter Notebook.	69
Rysunek 35. Przedstawienie interfejsu MatLab	70
Rysunek 36. Przedstawienie interfejsu repozytorium GitHub.	71
Rysunek 37. Fragment kodu odpowiedzialny za interakcje z użytkownikiem	72
Rysunek 38. Fragment kodu odpowiedzialny za generowanie krawędzi i wierzchołków	73
Rysunek 39. Fragment kodu odpowiedzialny za generowanie wag krawędzi	73
Rysunek 40. Fragment kodu odpowiedzialny za strumień pliku.....	73
Rysunek 41. Fragment odpowiedzialny za nazewnictwo wygenerowanego pliku	74
Rysunek 42. Proces działania programu po uruchomieniu	74
Rysunek 43. Informacje zwrocone przez program.....	74
Rysunek 44. Wizualizacja wygenerowanego dla 10 wierzchołków grafu	75
Rysunek 45. Deklaracja zmiennych globalnych oraz struktury Edge.....	76
Rysunek 46. Fragment kodu przedstawiający klasę comp_edge oraz strukturę vertex	76
Rysunek 47. Przedstawienie deklaracji oraz struktur w języku Python	77
Rysunek 48. Przedstawienie deklaracji oraz macierzy w Matlab	77
Rysunek 49. Fragment kodu języka C++ odpowiedzialny za wczytanie danych	78
Rysunek 50. Funkcja odpowiedzialna za wczytanie plików w języku Python	78
Rysunek 51. Wczytanie pliku w Matlab.....	79
Rysunek 52. Funkcja nocycle w języku C++	79
Rysunek 53. Przedstawienie funkcji nocycle() w języku Python.....	80
Rysunek 54. Funkcje sprawdzające występowanie cyklu w Matlab.....	80
Rysunek 55. Funkcja budująca MST w języku C++.....	81
Rysunek 56. Funkcja budująca MST w języku Python.....	82
Rysunek 57. Fragment kodu odpowiedzialny za budowę MST w Matlabie.....	82
Rysunek 58. Funkcja zapisująca dane do pliku CSV w języku C++.	83
Rysunek 59. Funkcja zapisująca dane do pliku CSV w języku Python.	83
Rysunek 60. Funkcja zapisująca dane do pliku CSV w Matlabie.	83
Rysunek 61. Implementacja funkcji main() w języku C++.....	84
Rysunek 62. Implementacja funkcji main() w języku Python.....	84
Rysunek 63. Funkcja mst() w Matlab.....	85
Rysunek 64. Efekt działania programu w C++, Pythonie oraz Matlabie.	85
Rysunek 65. Deklaracja zmiennych globalnych w języku C++.....	87
Rysunek 66. Deklaracja zmiennych globalnych w języku Python.....	87
Rysunek 67. Deklaracja zmiennej calculateInitialWeight w Matlab.	88
Rysunek 68. Przedstawienie funkcji Initialization() w języku C++.....	88

Rysunek 69. Przedstawienie funkcji Initialization() w języku Python.	89
Rysunek 70. Przedstawienie funkcji Initialization() w języku Matlabie.	90
Rysunek 71. Przedstawienie funkcji Calculations() w języku C++.	90
Rysunek 72. Przedstawienie funkcji Calculations() w języku Python.	91
Rysunek 73. Funkcja obliczeniowa Calculations przedstawiona w Matlabie.	92
Rysunek 74. Funkcja main() w języku C++.	93
Rysunek 75. Funkcja main() w języku Python.	94
Rysunek 76. Funkcja main() zaimplementowana w Matlabie.	95
Rysunek 77. Przedstawienie wyników działania programów C++ oraz Python.	95
Rysunek 78. Przedstawienie wyniku działania programu opracowanego w Matlabie.	96
Rysunek 79. Inicjalizacja zmiennych globalnych w języku C++.	97
Rysunek 80. Inicjalizacja klas SwapType oraz SwapData w języku Python	98
Rysunek 81. Funkcja SwapCheck w języku C++.	98
Rysunek 82. Klasa SwapCheck() w języku Python	99
Rysunek 83. Funkcja SwapCheck() przedstawiona w Matlabie.....	99
Rysunek 84. Funkcja Reverse() w języku C++.	100
Rysunek 85. Funkcja Reverse() w języku Python.	100
Rysunek 86. Funkcja Reverse() opracowana w Matlabie.....	101
Rysunek 87. Funkcja Initialization() w języku C++.	101
Rysunek 88. Funkcja Initialization() w języku Python.	102
Rysunek 89. Funkcja Initialization() w języku Matlabie.	102
Rysunek 90. Funkcja Calculations() w języku C++.	103
Rysunek 91. Funkcja Calculations() w języku Python.	104
Rysunek 92. Implementacja funkcji Calculations w Matlabie.	105
Rysunek 93. Implementacja funkcji SaveResultsToCSV w języku C++.	106
Rysunek 94. Implementacja funkcji SaveResultsToCSV w języku Python.	106
Rysunek 95. Implementacja funkcji SaveResultsToCSV w Matlabie.....	107
Rysunek 96. Implementacja funkcji głównej w języku C++.	107
Rysunek 97. Implementacja funkcji głównej w języku Python.	108
Rysunek 98. Implementacja funkcji głównej w Matlabie.....	108
Rysunek 99. Efekt działania programu w C++, Pythonie oraz Matlabie.....	109
Rysunek 100. Implementacja struktury Graph w języku C++.	110
Rysunek 101. Implementacja klasy Graph w języku Python.....	110
Rysunek 102. Implementacja struktury Graph w Matlabie	111
Rysunek 103. Implementacja algorytmu Prima w języku C++	112

Rysunek 104. Implementacja algorytmu Prima w języku Python.....	112
Rysunek 105. Implementacja algorytmu Prima w Matlabie.	113
Rysunek 106. Funkcja odnajdująca wierzchołki o nieparzystym stopniu w języku C++.	114
Rysunek 107. Funkcja odnajdująca wierzchołki o nieparzystym stopniu w języku Python..	114
Rysunek 108. Funkcja odnajdująca wierzchołki o nieparzystym stopniu w Matlabie.....	115
Rysunek 109. Implementacja funkcji minWeightMatching w języku C++.	115
Rysunek 110. Implementacja funkcji minWeightMatching w języku Python.....	116
Rysunek 111. Implementacja funkcji minWeightMatching w Matlabie.....	116
Rysunek 112. Fragment implementacji algorytmu Christofidesa w języku C++.....	117
Rysunek 113. Fragment implementacji algorytmu Christofidesa w języku Python.	118
Rysunek 114. Fragment implementacji algorytmu Christofidesa w Matlabie.	119
Rysunek 115. Dalsza część implementacji algorytmu Christofidesa w języku C++.	119
Rysunek 116. Dalsza część implementacji algorytmu Christofidesa w języku Python.	120
Rysunek 117. Dalsza część implementacji algorytmu Christofidesa w Matlabie.	120
Rysunek 118. Funkcja główna programu w języku C++.	121
Rysunek 119. Funkcja główna programu w języku Python.	122
Rysunek 120. Funkcja główna programu w Matlabie.	122
Rysunek 121. Funkcja saveResults() w języku C++.	123
Rysunek 122. Funkcja saveResults() w języku Python.	123
Rysunek 123. Funkcja saveResultsToCSV zaimplementowana w Matlabie.	123
Rysunek 124. Wyniki działania opracowanych programów	124
Rysunek 125. Wykres uzyskanych wyników dla algorytmu dwu-optymalnego w języku C++	
.....	135
Rysunek 126. Wykres uzyskanych wyników dla algorytmu trój-optymalnego w języku C++	
.....	137
Rysunek 127. Wykres uzyskanych wyników dla algorytmu Christofidesa w języku C++....	138
Rysunek 128. Wykres uzyskanych wyników dla algorytmu selekcji krawędzi w języku C++	
.....	140
Rysunek 129. Wykres uzyskanych wyników dla algorytmu dwu-optymalnego w Pythonie	141
Rysunek 130. Wykres uzyskanych wyników dla algorytmu trój-optymalnego w Pythonie..	142
Rysunek 131. Wykres uzyskanych wyników dla algorytmu Christofidesa w Pythonie.	143
Rysunek 132. Wykres uzyskanych wyników dla algorytmu selekcji krawędzi w Pythonie..	144
Rysunek 133. Wykres uzyskanych wyników dla algorytmu dwu-optymalnego w Matlabie.	145
Rysunek 134. Wykres uzyskanych wyników dla algorytmu trój-optymalnego w Matlabie..	146
Rysunek 135. Wykres uzyskanych wyników dla algorytmu Christofidesa w Matlabie.	148

Rysunek 136. Wykres uzyskanych wyników dla algorytmu selekcji krawędzi w Matlabie.	149
Rysunek 137. Wykres porównania czasów wykonania algorytmów w C++.....	150
Rysunek 138. Wykres porównania czasów wykonania algorytmów w Pythonie.....	151
Rysunek 139. Wykres porównania czasów wykonania algorytmów w Matlabie.....	152
Rysunek 140. Wykres złożoności obliczeniowej algorytmów	153
Rysunek 141. Analiza całkowitej wagi trasy algorytmów w zależności od liczby wierzchołków	154

Spis tabel

Tabela 1	22
Tabela 2	54
Tabela 3	63
Tabela 4	63
Tabela 5	64
Tabela 6	64
Tabela 7	65
Tabela 8	126
Tabela 9	127
Tabela 10	128
Tabela 11	129
Tabela 12	130
Tabela 13	131
Tabela 14	132
Tabela 15	133
Tabela 16	134
Tabela 17	153

Spis załączników

1. Płyta CD zawierająca:

- Praca-magisterska-Rzepiela-Filip.docx – praca dyplomowa w formacie docx.
- Praca-magisterska-Rzepiela-Filip.pdf – praca dyplomowa w formacie pdf.
- Programy-Python.zip – programy algorytmów zaimplementowane w języku Python.
- Programy-Matlab.zip – programy algorytmów zaimplementowane za pomocą programu Matlab.
- Programy-Cpp.zip – programy algorytmów zaimplementowane w języku C++
- Generator.zip – generator grafów losowych zaimplementowany w języku C++
- Dane_wejsciowe.zip – dane, na których przeprowadzone zostały badania