AGH
Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Klaudia Fil

kierunek studiów: **informatyka stosowana**

Problem chińskiego listonosza w sieci ulic w Krakowie

Opiekun: **dr hab. inż. Przemysław Gawroński**

Kraków, styczeń 2021

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

.....

(czytelny podpis)

Ocena merytoryczna opiekuna

Ocena merytoryczna recenzenta

Spis treści

1 Wstęp

1.1 Wprowadzenie

Szeroko pojęty problem związany z wyznaczaniem trasy (ang. *General Routing Problem*) zdefiniowano jako szukanie drogi o minimalnym koszcie, która dodatkowo musi spełniać uwzględnione w planowaniu wymagania[1]. Skupiając się na praktycznym aspekcie GRP należy wziąć pod uwagę jeden z bardziej znanych przypadków rozważań - problem chińskiego listonosza (ang. *Chinese Postman Problem*)[2].

W życiu codziennym wiele zawodów związanych jest wyznaczaniem trasy na tle procesów logistycznych, wśród nich różnego rodzaju dostawca, kurier czy właśnie listonosz. Ich szlakiem docelowym jest droga zawierająca w sobie każdą ulicę danego obszaru przynajmniej raz. Optymalnym rozwiązaniem CPP byłoby uniknięcie ponownych przejść jedną ścieżką, jednak w sytuacjach rzeczywistych jest to często niemożliwe.

1.2 Cel pracy

Celem pracy dyplomowej jest zaimplementowanie dwóch algorytmów, rozwiązujących kwestię znajdowania pełnej ścieżki z cyklem zamkniętym na grafie spójnym, przy optymalizacji kosztów przemieszczania się, co sprowadza się do CPP. Rozpatrzono w niej przypadki rzeczywiste, gdzie w powstałych z mapy Krakowa grafach mieszanych (ang. *Mixed Graph*) (nazywanych również ogólnie jako grafy skierowane (ang. *Directed Graph*)) oraz nieskierowanych (ang. *Undirected Graph*), znajdowana jest najlepsza droga dla listonosza[3].

Jeden z algorytmów przedstawiony w pracy dedykowany jest MG. Swoje zastosowanie ma, kiedy przebieg trasy listonosza uwarunkowany jest czynnikami zewnętrznymi tj. drogami jedno- lub dwukierunkowymi. W przypadku pieszego ruchu, w którym sposób przejścia jest dowolny, problem dotyczy

UG. Zaimplementowano dla niego drugi z schematów umożliwiający wyznaczenie najbardziej efektywnej trasy. Graf rzeczywisty tworzony jest z małych fragmentów miasta, dlatego założono, że poruszać się będzie na nim tylko jeden listonosz.

Dla sprawdzenia poprawności oraz wyznaczenia złożoności algorytmów wygenerowano losowe sieci, które umożliwiły szersze ich przetestowanie. Stworzono również program do wizualizacji grafów, oraz kolejnych etapów ścieżek, umożliwiający graficzne prześledzenie działania schematów.

1.3 Problemu chińskiego listonosza

Problem chińskiego listonosza jest jedynym z podstawowych zagadnień związanych z wyznaczaniem trasy. Swoją nazwę zawdzięcza chińskiemu matematykowi, który jako pierwszy go sformułował. Analizował drogę dostawcy w jednym z chińskich miast, który miał dostarczyć wszystkie przesyłki, odwiedzając każdą ulice, nie nadkładając niepotrzebnie drogi i powrócić do bazy[4].

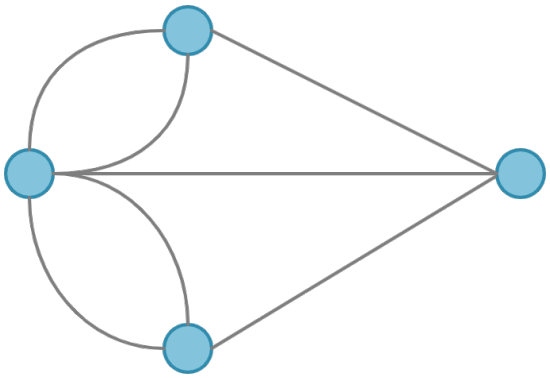
Patrząc przez pryzmat przypadku rzeczywistego, gdzie rozważany jest listonosz, czy dostawca ulotek, CPP sprowadza się do procesu wyboru najlepszej ścieżki w sieci dróg, gdzie każda ulica musi zostać odwiedzona przynajmniej raz, czyli odnalezienia na grafie cyklu Eulera[5].

1.4 Cykl Eulera

Cykl Eulera (ang. *Eulerian Cycle*) jest to ścieżka poprowadzona na dowolnym grafie, która przechodzi przez każdą z krawędzi dokładnie raz. Nazywamy ją cyklem, ponieważ zaczyna się i kończy w tym samym wierzchołku.

Nazwa takiego przejścia przez graf nadana została na cześć matematyka Leonharda Eulera, który w swojej pracy *Solutio problematis ad geometriam situs pertinentis*[6] poruszył zagadnienie mostów królewieckich. Odpowiadał

na pytanie, czy istnieje ścieżka prowadząca przez wszystkie mosty, biorąc pod uwagę, że przez każdy z nich można iść tylko raz.



Rysunek 1: Graf przestawiający problem mostów w Królewcu

Mapa, którą rozważał, sprowadzała się do grafu z rysunku 1, gdzie krawędzie odpowiadają siedmiu mostom. Wykazał, że znalezienie EC jest możliwe tylko dla grafów spełniających określone warunki, nazwanych na jego cześć grafami Eulera. Była to jedna z pierwszych prac związanych z teorią grafów.

1.4.1 Graf Eulera

Twierdzenie 1 (Euler, 1736) *Graf spójny jest eulerowski wtedy i tylko wtedy, gdy stopień każdego wierzchołka jest liczbą parzystą.*

Twierdzenie 2 *Graf skierowany spójny zawiera cykl Eulera wtedy i tylko wtedy, gdy dla każdego wierzchołka v zachodzi $d^+(v) = d^-(v)$, gdzie*

$d^+(v)$ - ilość krawędzi wchodzących do wierzchołka v,

$d^-(v)$ - ilość krawędzi wychodzących z wierzchołka v.

Aby możliwe było znalezienie poprawnej ścieżki dzięki opisanym w tej pracy algorytmom, grafy w nich wykorzystywane muszą spełniać twierdzenia 1 i 2.

1.5 Sieci złożone

1.5.1 Sieć Barabási–Albert

Pod koniec XX w. na Uniwersytecie Notre Dame w Indianie Reka Albert i Albert-Laszlo Barabasi podczas badania struktury sieci WWW przypadkiem odkryli ciekawą zależność, a mianowicie potęgowe rozkłady prawdopodobieństwa opisujące połączenia pomiędzy stronami internetowymi.

Rozkład ten wyglądał następująco:

$$P(k) \sim k^{-\alpha} \tag{1}$$

co oznaczało, że sieć WWW ma własności fraktalne.

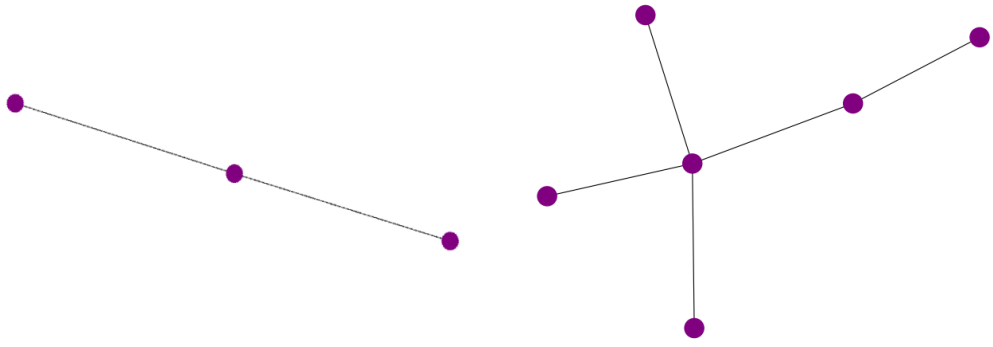
Zauważono, że większość układów rzeczywistych również charakteryzuje się rozkładem podobnym do równania 1. Szerszą analizę i reguły tworzenia sieci tego typu panowie Albert i Barabasi przedstawili w swojej pracy zatytułowanej *Emergence of scaling in random networks*[7]. Ustalili, że rozkład 1 wynika z wzrostu sieci i reguły związanej z preferencyjnym dołączaniem węzłów. Model o takich zależnościach i rozkładzie nazwano na ich część.

Formułowanie ewoluującej sieci polega na stopniowym jej rozroście w każdym momencie czasowym. Proces rozpoczęty w chwili zerowej składa się z grafu zawierającego m_0 połączonych wierzchołków. Podczas kolejnych kroków czasowych nowo dodawany węzeł łączony jest z m innymi, istniejącymi już wierzchołkami. Szansa, że świeżo utworzony wierzchołek połączony zostanie krawędzią do starego węzła jest proporcjonalne do stopnia węzła k_i:

$$\Pi(k_i) = \frac{k_i}{\sum_{i=1}^{t} k_i} \tag{2}$$

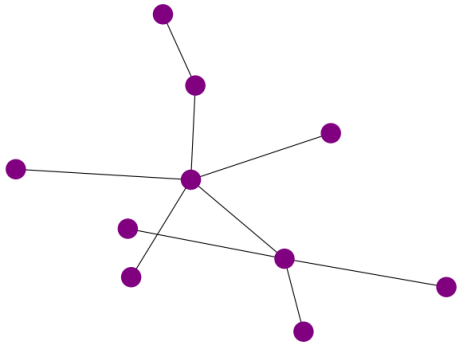
co przedstawione jest na rysunku 2.

Algorytm posiłkowy, z którego skorzystano w pracy, tworzący grafy w wyżej opisanym modelu to *barabasi_albert_graph*, zaczerpnięto go z biblioteki *Networkx* szerzej opisanej w rozdziale 2.

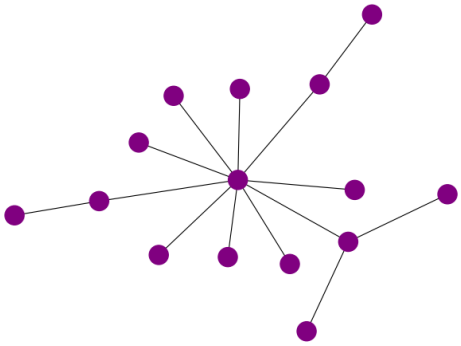


(a) $m_0 = 1$, $t = 3$

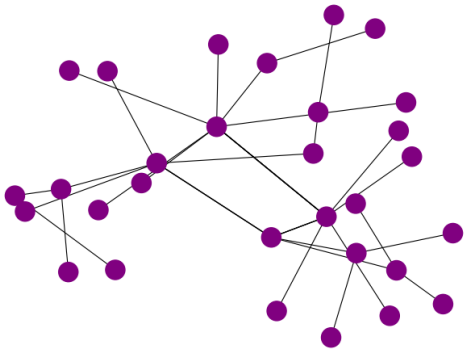
(b) $m_0 = 1$, $t = 6$



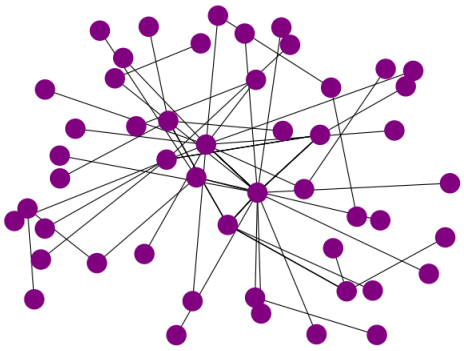
(c) $m_0 = 1$, $t = 10$



(d) $m_0 = 1$, $t = 15$



(e) $m_0 = 1$, $t = 30$



(f) $m_0 = 1$, $t = 50$

Rysunek 2: Graf losowy stworzony zgodnie z modelem Barabási–Albert o początkowych wierzchołkach m_0 w kolejnych krokach czasowych t.

1.5.2 Sieć Wattsa–Strogatza

W 1967 r. psycholog Stanley Milgram przeprowadził eksperyment *Small-world project*[8], który opisuje połączenia w dużej społeczności. Zaobserwował on, że większość osób potrzebuje niewielkiej ilości przyjaciół, którzy mogą przedstawić ich swoim znajomym, umożliwiając poznanie całej badanej grupy. Wyniki eksperymentu nazwano *hipotezą o małym świecie*, co oznacza, że każdego człowieka łączy stosunkowo niewielka liczba pośredników w danym społeczeństwie.

Charakterystyka modelu grafu, którego konsekwencja to zjawisko zaobserwowanego przez Milgrama, została zaproponowana w 1998 r. przez Duncana Wattsa i Stephena Strogatza w *Collective dynamics of 'small-world' networks*[9], na których część nazwano topologię sieci z opisanymi przez nich typami połączeń.

Model strukturalny powstaje z sieci regularnej, w której wprowadzona zostaje randomizacja wierzchołków. Ponadto, aby został zakwalifikowany do kategorii *small-world*, jej parametry: współczynnik sklastrowania ${C_g}^{\Delta}$ zadany wzorem

$$ {C_g}^{\Delta} = \frac{3t}{k}, \tag{3} $$

gdzie:

t - liczba węzłów tworzących *trójkąty*, czyli trzech wierzchołków połączonych każdy z każdym,

k - liczba ścieżek o długości 2,

oraz średnia odległość pomiędzy dowolną parą wierzchołków L_g muszą spełniać wymogi twierdzenia 3[10].

Twierdzenie 3 *O sieci G mówi się, że jest siecią typu mały świat, jeżeli* $L_g \geq L_{rand}$ *oraz* ${C_g}^{\Delta} \gg {C_{rand}}^{\Delta}$,
gdzie

L_g *- najkrótsza ścieżka grafu G,*

L_{rand} *- najkrótsza ścieżka pomiędzy dwoma węzłami losowego grafu, stworzonego z takiej samej ilości wierzchołków i krawędzi jak graf G, zachowując przy tym prawdopodobieństwo przypisania krawędzi do pary węzłów,*

${C_g}^{\Delta}$ *- współczynnik klastrowania grafu G, wyznaczony ze wzoru 3,*

${C_{rand}}^{\Delta}$ *- współczynnik klastrowania grafu losowego, stworzonego z takiej samej ilości wierzchołków i krawędzi jak graf G, zachowując przy tym prawdopodobieństwo przypisania krawędzi do pary węzłów.*

Wykorzystując twierdzenie 3 do wyprowadzenia zależności:

$$ {\gamma_g}^{\Delta} = \frac{C_g^{\Delta}}{C_{rand}^{\Delta}}, \tag{4} $$

$$ \lambda_g = \frac{L_g}{L_{rand}}, \tag{5} $$

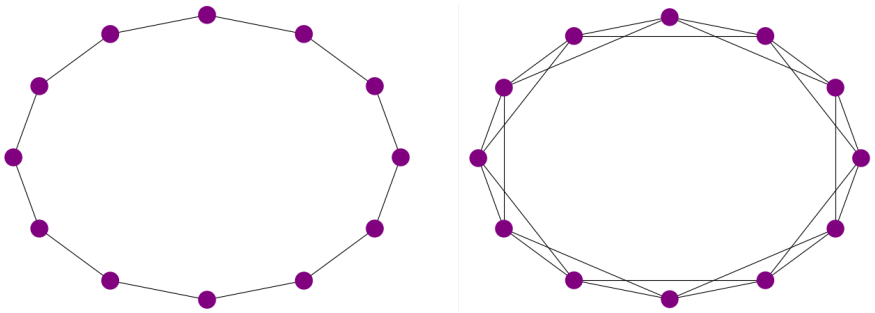
sformułowano końcowy postulat:

Twierdzenie 4 *O sieci G mówi się, że jest siecią małego świata, jeżeli* $S^{\Delta} > 1$,
gdzie

$S^{\Delta} = \frac{{\gamma_g}^{\Delta}}{\lambda_g}$.

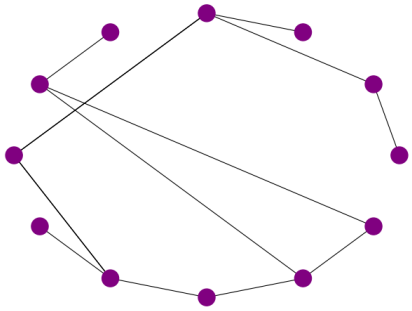
Współczynnik sklastrowania określa stopnień wykorzystania wszystkich możliwych połączeń międzywęzłowych. C w praktyce pozwala na ocenienie stopnia oddalenia pomiędzy wierzchołkami[11].

Algorytm generujący grafy o modelu Wattsa–Strogatza wykorzystany w pracy to *watts_strogatz_graph*, który podobnie jak model 1.5.1 pochodzi z biblioteki *Networkx*.

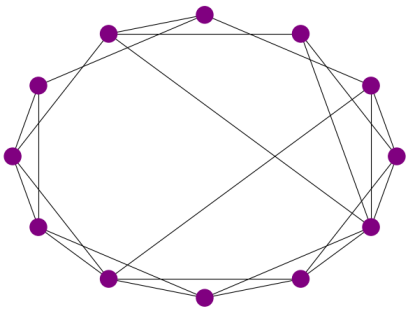


(a) $n = 12, k = 2, p = 0$

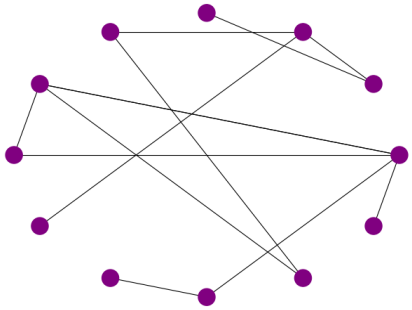
(b) $n = 12, k = 4, p = 0$



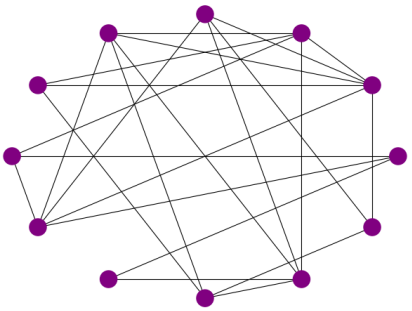
(c) $n = 12, k = 2, p = 0.3$



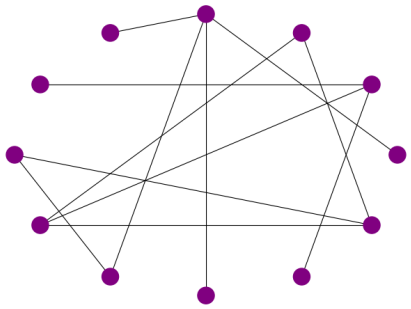
(d) $n = 12, k = 4, p = 0.3$



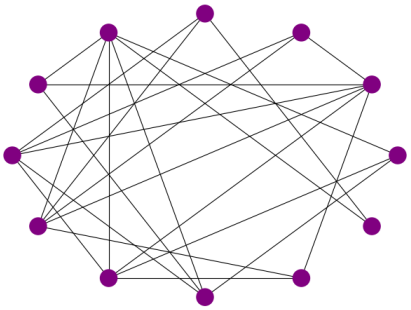
(e) $n = 12, k = 2, p = 0.6$



(f) $n = 12, k = 4, p = 0.6$



(g) $n = 12, k = 2, p = 1$



(h) $n = 12, k = 4, p = 1$

Rysunek 3: Sieć stworzona zgodnie z modelem Wattsa–Strogatza, gdzie n - liczba wierzchołków, k - ilość najbliższych sąsiadów do połączenia w topologii pierścienia, p - prawdopodobieństwo zmiany połączenia.

Przy tworzeniu losowych grafów zgodnie z modelem Wattsa-Strogatza posłużono się wartościami n, k, p, które kolejno oznaczają ilość wierzchołków, ilość połączeń pomiędzy najbliższymi sąsiadami, prawdopodobieństwo zmiany jednego z węzłów krawędzi.

Na rysunku 3 przedstawiono wygląd randomizacji wierzchołków w modelu sieci 12-to węzłowej dla dwóch przypadków ilości połączeń: dwóch sąsiadów na 3a, 3c, 3e, 3g oraz czterech na 3b, 3d, 3f, 3h, gdzie każdy kolejny rysunek prezentuje wygląd po losowym przydzieleniu nowych krawędzi dla prawdopodobieństw $p = 0, 0.3, 0.6, 1$, gdzie $p \in [0, 1]$. Wraz z wzrostem p zauważalne jest odejście od początkowej regularności układu przy zerowej wartości. Dla górnego przedziału p sieć praktycznie wcale nie przypomina modelu *small-world*, czego można się spodziewać po odstępstwie od założeń zgodnych z twierdzeniami 3 i 4.

2 Narzędzia

Aplikacja w całości została zaimplementowana w Pythonie[12], ze względu na jego prostotę i czytelność, które ułatwiały znacznie pracę. Innym atrybutem przemawiającym za tym językiem jest duża ilość bibliotek związanych z teorią grafów. Jedna z nich została wykorzystana przy tworzeniu projektu, m. in. do wyliczania najkrótszych ścieżek, ale również przy tworzeniu losowych grafów.

Biblioteki użyte w pracy:

- *Networkx*[13] - biblioteka w języku Python używana przy badaniu i tworzeniu dużych grafów i sieci. Wykorzystana została w algorytmie konwertującym graf do takiego, który posiada ścieżkę Eulera opisaną w 1.4 i podczas tworzenia przypadkowego digrafu oraz sieci złożonych z podpunktu 1.5.

- *Matplotlib*[14] - jedna z najbogatszych bibliotek do tworzenia wykresów w języku Python. W pracy gównie wykorzystano zawarte w niej API *pylab* wykorzystujące prosty interface analogiczny do środowiska MATLAB[15].

Inną istotną rzeczą użytą w projekcie to mapa świata OpenStreetMap (OSM)[16], która zbudowana jest z danych gromadzonych przez szeroką społeczność. Dowolna osoba może przyczynić się do jej rozwoju poprzez dostarczenie informacji geograficznych zbieranych przez urządzenia z odbiornikami GPS lub zdjęcia satelitarne. Dane w niej są udostępnione za darmo do ogólnego użytku.

3 Algorytmy i implementacja

3.1 Modyfikacja do grafu Eulera

Algorytmy Fleury'ego i Hierholzera mające wyznaczyć ścieżki zakładają, że obiekty, które przeszukują, są grafami Eulera, co opisano szerzej w rozdziale 1.4. Ponieważ dane testowe każdego rodzaju (losowy DG, sieci o modelu Barabásiego–Alberta oraz Wattsa–Strogatza) tworzone są randomowo, przez funkcje biblioteczne *Networkx*, wymagana jest ich modyfikacja, gdyby nie było w nich cyklu Eulera. Na potrzeby tych warunków przeprowadzono implementacje dwóch algorytmów.

3.1.1 Modyfikacja grafu nieskierowanego

W przypadku wygenerowania UG sprawdzane jest pierwsze, czy modyfikacje są wymagane, realizowane jest to przy pomocy funkcji *is_eulerian* z biblioteki grafowej opisanej w rozdziale 2. Kiedy zwracana jest informacja o niezgodności z twierdzeniem 1 wywoływana jest funkcja odpowiedzialna za dodanie duplikatów krawędzi, które zapewnią poprawny stopnień każdego z wierzchołków.

Początkowo uzupełniana jest lista z nieparzystymi wierzchołkami (linie 4-6 alg. 1). Ich nieparzysta ilość uniemożliwia stworzenie połączeń, zwracana jest wartość *None*, która wywołuje mechanizm odpowiedzialny za ponowne wylosowanie wartości, aż modyfikacja będzie możliwa.

Algorytm 1: *makeEulerianGraph* przekształcający graf nieskierowany do grafu Eulera

```
1  def makeEulerianGraph(graph):
2          listOfOddDegreeNodes = []
3
4          for node in graph.nodes(data=True):
5                  if idOddNumber(graph.degree[node[0]]):
6                          listOfOddDegreeNodes.append(node[0])
7          if idOddNumber(len(listOfOddDegreeNodes)):
8                  return None
9
```

```
10          listOfOddDegreeNodes1 = listOfOddDegreeNodes[:len(listOfOddDegreeNodes) // 2]
11          listOfOddDegreeNodes2 = listOfOddDegreeNodes[len(listOfOddDegreeNodes) // 2:]
12          listOfRandomParams = list()
13          for x in range(0, 50):
14                  random.shuffle(listOfOddDegreeNodes2)
15                  listOfRandomParams.append(list(zip(listOfOddDegreeNodes1,
16                                                     listOfOddDegreeNodes2)))
17          minPath = getOptimalAdditionalPaths(graph, listOfRandomParams)
18          appendFakeEdges(graph, minPath)
```

W idealnych warunkach wierzchołki o nieparzystych stopniach dobierane są w możliwie najlepsze pary poprzed funckję *getOptimalAdditionalPaths* (wykorzystującą algorytm najkrótszych ścieżek z biblioteki *Networkx*) i łączone ze sobą. Niestety dla większych grafów liczby par są bardzo duże, a ilość ich możliwych permutacji nie pozwalają na modyfikacje sieci w rozsądnym czasie. Zdecydowano się na ograniczenie wywołań funkcji bibliotecznej z linii 7 algorytmu 2, poprzez obliczenie ścieżek tylko dla części możliwych par, których wybranie zaprezentowane jest w liniach 10-16 *makeEulerianGraph* (alg. 1) Funkcje znajdowania najkrótszych ścieżek wykorzystują wagi UG. Wszystkie grafy generowane do testowania posiadają losowo przydzielone wagi z przedziału $[1, 10]$, do każdej krawędzi.

Algorytm 2: *getOptimalAdditionalPaths* funkcja obliczająca i zwracająca najlepsze pary wierzchołków do połączenia, dla *makeEulerianGraph* i *makeEulerianDiGraph*

```
1  def getOptimalAdditionalPaths(graph, listWithAllPerm):
2          minDist = float('inf')
3          minPath = []
4          for currentList in listWithAllPerm:
5                  dist = 0
6                  for pair in currentList:
7                          dist += nx.shortest_path_length(graph, source=pair[0],
8                          target=pair[1], weight='weight')
9                  if dist < minDist:
10                         minDist = dist
11                         minPath = currentList
12         return minPath
```

Finalnie po znalezieniu możliwie najlepszych dodatkowych połączeń, są one dodawane do wejściowego grafu (linia 20, alg. 1) z odpowiednią etykietą informującą, że jest to jedna z powielonych krawędzi.

3.1.2 Modyfikacja grafu skierowanego

Analogiczna sytuacja do 3.1.1 ma miejsce dla DG w odniesieniu do twierdzenia 2. Początkowe informacje klasyfikujące wierzchołek do dodania nowych krawędzi zbierane są w liniach 2-16 algorytmu 3, gdzie sprawdzana jest różnica połączeń wchodzących a wychodzących. Następnie również odwołujemy się do algorytmu 2, przystosowanego zarówno dla UG i DG, tym razem wysyłając tablice *negNodes* - wierzchołki z liczniejszą ilością krawędzi wchodzących oraz *posNodes* - węzły z większą liczbą wyjść. I w tym przypadku optymalnie jest zrezygnować z wszystkich permutacji. Po obliczeniach dodawane są fałszywe krawędzie z odpowiednią etykietą do grafu.

Algorytm 3: *makeEulerianDiGraph* przekształcający graf skierowany do grafu Eulera

```
1  def makeEulerianDiGraph(graph):
2          diff = [None] * len(
3                          graph.nodes)  # list with nodes difference :
4                                        # predecessors nodes - successors nodes
5          for node in graph.nodes(data=True):
6                  diff[node[0]] = len(graph.pred[node[0]]) - len(graph.succ[node[0]])
7
8          negNodes = []
9          posNodes = []
10         for node in range(0, len(diff)):
11                 if diff[node] < 0:
12                         for rep in range(-1 * diff[node]):
13                                 negNodes.append(node)
14                 if diff[node] > 0:
15                         for rep in range(diff[node]):
16                                 posNodes.append(node)
17
18         if idOddNumber(len(negNodes) - len(posNodes)):
19                 return None
20
21         listOfRandomParams = list()
22
23         for x in range(0, 50):
24                 random.shuffle(negNodes)
25                 listOfRandomParams.append(list(zip(posNodes, negNodes)))
26
27         minPath = getOptimalAdditionalPaths(graph, listOfRandomParams)
28         appendFakeEdges(graph, minPath)
```

3.2 Graf rzeczywisty

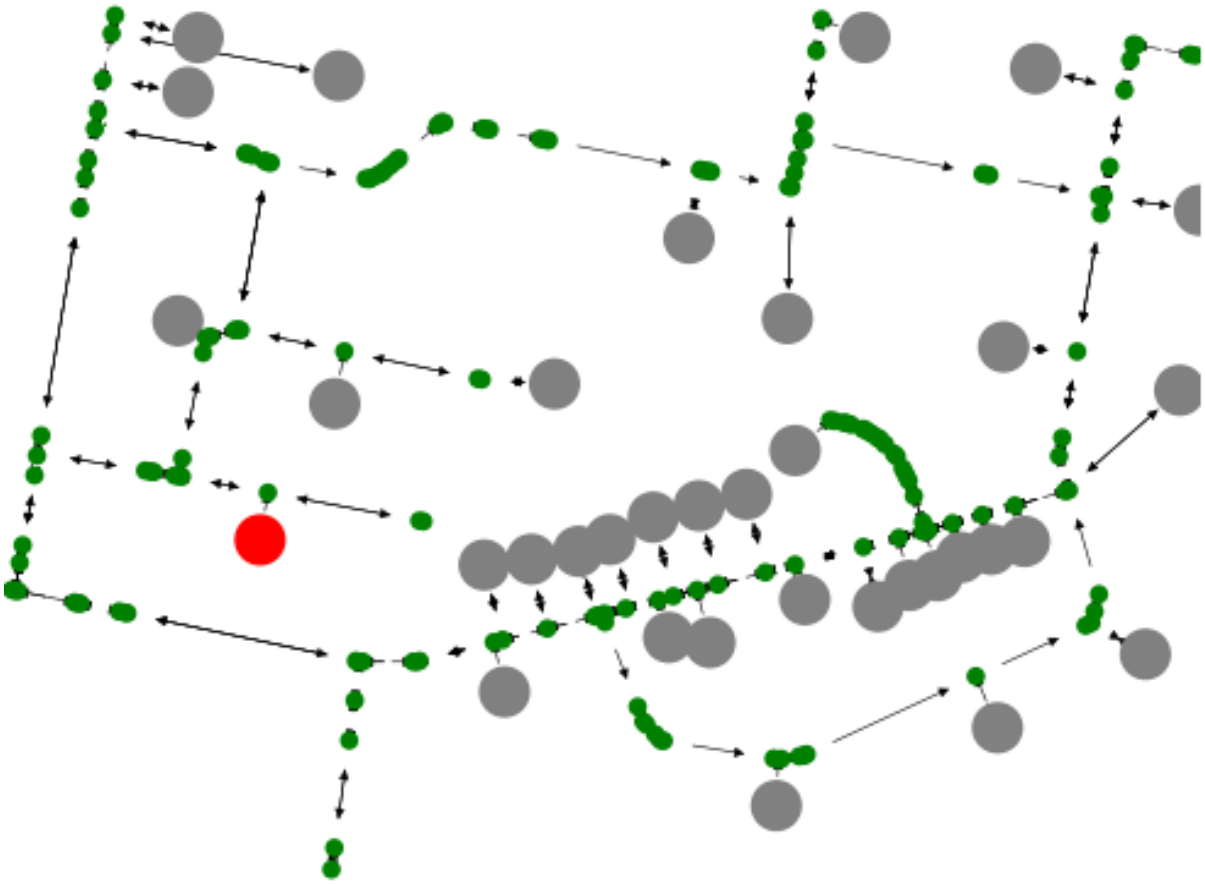
Algorytmy docelowo szukają optymalnej trasy na planie miasta Kraków. Poprawne ich zastosowanie wymaga konwersji mapy na postać grafową.

Rysunek 4: Fragment mapy Krakowa[1]

Fragment programu przetwarzający mapę z rysunku 4 formuje graf, którego krawędzie odpowiadają ulicom, a wierzchołki reprezentują zarówno budynki, jak i istotne elementy dróg: skrzyżowania i strategiczne punkty trasy, które to potrzebne są do idealnego przeprowadzenia ulicy, czyli uwzględnienia zakrętów lub nieliniowych fragmentów drogi.

Program odpowiedzialny za generowanie grafu przy pobieraniu danych z bazy OpenStreetMap filtruje odpowiednie informacje o obiektach i znajduje placówkę pocztową (o ile taka istnieje na danych fragmencie miasta) dzięki konkretnym tagom[17]. W przypadku wgrania fragmentu mapy, gdzie

[1] Zrzut ekranu z strony https://www.openstreetmap.org/



Rysunek 5: Graf rzeczywisty(mieszany) stworzony z mapy z rysunku 4, gdzie czerwone węzeł oznaczają początek trasy listonosza, szare są budynkami mieszkalnymi, zielone to fragmenty ulicy zachowane w celu odwzorowania rzeczywistego jej przebiegu tj. zakrętów, skrzyżowań. Kierunki ulic rzeczywistych (jedno- lub dwukierunkowe drogi) mają wpływ na obecność i kierunek krawędzi.

nie ma poczty, obierany jest inny punkt początkowy trasy listonosza, czy sprzedawcy, który losowany jest wśród istniejących na planie budynków.

Wierzchołki w grafie umieszczono na rysunku odwzorowując rzeczywiste położenie na mapie, przez co krawędzie stworzone na drogach, czy pomiędzy ulicami a budynkami mają odpowiednio proporcjonalne długości. Węzły oznaczono różnymi kolorami, w celu zwiększenia czytelności. Placówkę pocztową lub punkt startowy przedstawiono na czerwono, zwykłe budynki mieszkalne na szaro. Natomiast wierzchołki będące fragmentami drogi poko-

lorowano na zielono i zmniejszono ich rozmiar względem pozostałych.

Analizując kierunki dróg na rysunku 4 można zauważyć, że wpływają one na budowę grafu rzeczywistego z rysunku 5, który w przypadku kiedy listonosz porusza się np. samochodem jest grafem mieszanym. Niektóre z krawędzi mogą być słabo widoczne, ze względu na duże zagęszczenie węzłów.

3.2.1 Implementacja grafu rzeczywistego

Graf rzeczywisty stworzony został jako obiekt klasy *OsmGraph* przy pomocy funkcji z *OsmParser*, która to odpowiedzialna była za poprawne sparsowanie łańcucha znaków danych w formacie XML[18].

Dane pobierane z strony internetowej OpenStreetMap za pomocą algorytmu 4.

Algorytm 4: *getOsmData* funkcja pobierająca dane z strony *opensteermap.org* w formacie *osm*, dla zadanych jej początkowych wartości szerokości i długości geograficznych, tworzących prostokątny obszar do wygenerowania danych.

```
1   def getOsmData(top, right, bottom, left):
2           request = "http://api.openstreetmap.org/api/0.6/map?bbox=%f,%f,%f,%f"
3                                                   % (left, bottom, right, top)
4           fp = urllib.request.urlopen(request)
5           return fp.read().decode('utf-8')
```

Zaimplementowane zostało to tak, żeby główna klasa odpowiedzialna za przechowywanie danych w formacie grafu, na którym mogłyby być wykonywanie algorytmy szukające cykli Eulera, czyli *OsmGraph*, posiadała obiekt *MultiDiGraph* z biblioteki *Networkx*. Wszystkie funkcje z tej klasy operują na nim, odpowiednio modyfikując jego strukturę przy dostosowaniu go rzeczywistości.

Przy przetwarzaniu danych z XML wykorzystano dwie pomocnicze struktury $OsmNode$ i $OsmWay$, odpowiedzialne za przechowanie podstawowych informacji wyciągniętych z bazy OSM.

Algorytm 5: *OsmNode* struktura tagu XML - *node*, w którym przechowywane są punkty

```
class OsmNode(object):
        def __init__(self, id, x, y):
                self.id = id
                self.x = x
                self.y = y
                self.tags = {}
```

Algorytm 6: *OsmWay* struktura tagu XML - *node*, w którym przechowywane są punkty.

```
        class OsmWay(object):
                def __init__(self, id, osm):
                        self.id = id
                        self.osm = osm
                        self.nds = []  # list of nodes' references
                        self.tags = {}
```

Kolejne elementy zaimplementowane do stworzenia grafu rzeczywistego:

1. Pobranie danych przy pomocy klasy *OsmParser*, które przechowywane są w listach jako obiekty typu alg. 5 i alg. 6.

2. Wyszukanie budynku z listy $OsmWay$ i $OsmNode$, który w swoich tagach zawiera informację, czy jest to placówka pocztowa. Ewentualne losowe wybranie punktu startowego wśród budynków.

3. Dodanie budynków mieszkalnych jako wierzchołków grafu.

4. Wyznaczenie potencjalnych ulic posługując się obiektami typu $OsmNode$ przechowywanymi jako referencje w polu klasy $OsmWay$ *nds* oraz podstawowego równania na prostą o dwóch zadanych punktach (kolejnych referencjach z listy).

5. Połączenie budynków i potencjalnych punktów ulic, obliczając minimalne odległości wykorzystując funkcję z algorytmu 7.

6. Wykonanie funkcji modyfikującej do grafu Eulera opisanej w punkcie 3.1.2. W przypadku, kiedy obszar zaznaczony jest niemożliwy do konwersji(część ulicy jednokierunkowej urwanej na krawędzi), program koń-

czy działanie z informacją, że dla tego fragmentu mapy nie jest w stanie wyznaczyć pożądanej ścieżki.

Odległości pomiędzy budynkami zostały zachowane jako wartości wag odpowiednich krawędzi. Ponieważ pobrano informacje o położeniu geograficznym wskazanych budynków na globie, wykorzystano funkcję Haversine[19], do wyznaczenia dystansu (równania 6 - 8) między nimi. Określa ona najkrótszą odległość po ortodromie pomiędzy dowolnymi punktami na globie, dzięki ich współrzędnym geograficznym.

$$a = \sqrt{sin^2(\frac{\Delta\phi}{2}) + cos(\phi_1) \cdot cos(\phi_2) \cdot sin^2(\frac{\Delta\lambda}{2})} \tag{6}$$

$$c = 2 \cdot asin(\sqrt{a}) \tag{7}$$

$$d = R \cdot c \tag{8}$$

Algorytm 7: *getDistance* funkcja obliczająca odległości między rzeczywistymi budynkami, generująca wagi dla krawędzi. Przy jej implementacji sugerowano się wzorami 6 - 8.

```
1  def getDistance(x1, y1, x2, y2):
2          x1, y1, x2, y2 = map(radians, [x1, y1, x2, y2])
3          dx = x2 - x1
4          dy = y2 - y1
5          a = sin(dy / 2) ** 2 + cos(y1) * cos(y2) * sin(dx / 2) ** 2
6          c = 2 * asin(sqrt(a))
7          r = 6371
8          r *= 1000        #metres
9          return c * r
```

3.3 Algorytm Fleury'ego

Jednym z prostszych sposobów znajdowania cyklu Eulera w UG jest algorytm Fleury'ego. Przede wszystkim zakłada on, że graf, który przyjmuje jest eulerowski, co zapewnia algorytm 1.

Algorytm 8: *FleuryAlgorithm* wyszukujący ścieżkę w grafie nieskierowanym

```
1        def FleuryAlgorithm(graph, startNode):
2        adjList = GraphHelper.getAdjList(graph)
3        EulerCycle = list()
4
5        findNextNode(adjList, startNode, EulerCycle)
6        return EulerCycle
```

UG, na którym wywołujemy *FleuryAlgorithm* konwertowany przez prostą funkcję *getAdjList* do listy sąsiedztwa, jednej z możliwych reprezentacji grafu, która zazwyczaj jest jego najefektywniejszym przedstawicielstwem. Wierzchołek startowy, narzucony przez parametr wejściowy, jest tym od którego zaczynane jest poszukiwanie ścieżki. Algorytm odwiedza kolejne krawędzie i usuwa je odkładając kolejno na listę, kierując się zasadą, że połączenie będące mostem (czyli taką krawędzią w grafie, której usunięcie zwiększy ilość jego spójnych składowych SS[2]) usuwane jest w ostateczności. Program trwa, aż do odwiedzenia wszystkich krawędzi, a cykl Eulera, będącym rozwiązaniem algorytmu, jest listą kolejno odwiedzonych wierzchołków.

Algorytm 9: *findNextNode* rekurencyjna funkcja pomocnicza dla *FleuryAlgorithm*

```
1        def findNextNode(adjList, n, EulerCycle):
2        EulerCycle.append(n)
3        for v in adjList[n]:
4        if isBridge(adjList, n, v):
5        removeEdge(adjList, n, v)
6        findNextNode(adjList, v, EulerCycle)
```

Implementacja problemu w pracy została podzielona na 3 główne funkcje: *FleuryAlgorithm* przedstawiony jako nr 8, który wywołuje rekurencyjną

[2]Jako spójną składową określamy taki podgraf grafu, który jesteśmy w stanie wyodrębnić z całości bez usuwania krawędzi

część oznaczoną jako algorytm 9 oraz fragment sprawdzający, czy aktualnie sprawdzana krawędź jest mostem (alg. 10).

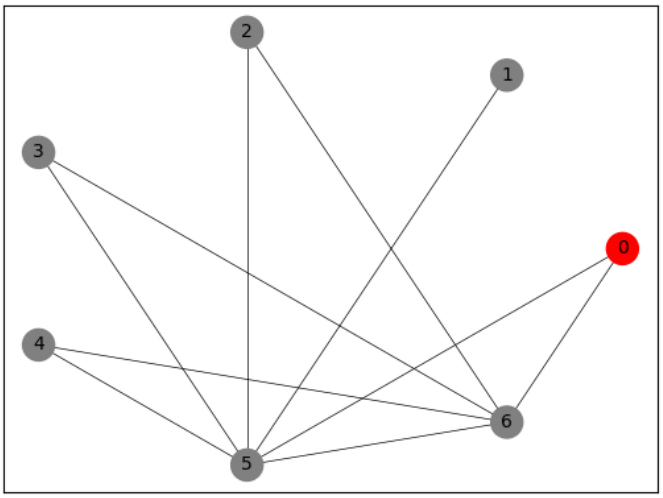
W algorytmie 10, w przypadku gdy ilość sąsiadów jest większa niż jeden, program najpierw posiłkuje się prostym algorytmem rekurencyjnym DFS (przeszukiwanie w głąb[20]), który ma za zadanie wyznaczyć ilość spójnych składowych. Następnie usuwana jest sprawdzana krawędź i ponownie obliczana jest ilość SS. Finalnie przywraca się wcześniej wspomniane połączenie między-wierzchołkowe. W zależności, czy wartości obliczone są różne zwracana jest zmienna typu $Bool$, pozwalająca lub nie na poszukiwanie kolejnego wierzchołka w funkcji *findNextNode*.

Złożoność obliczeniowa szacowana jest na $O(V(V+E))$.

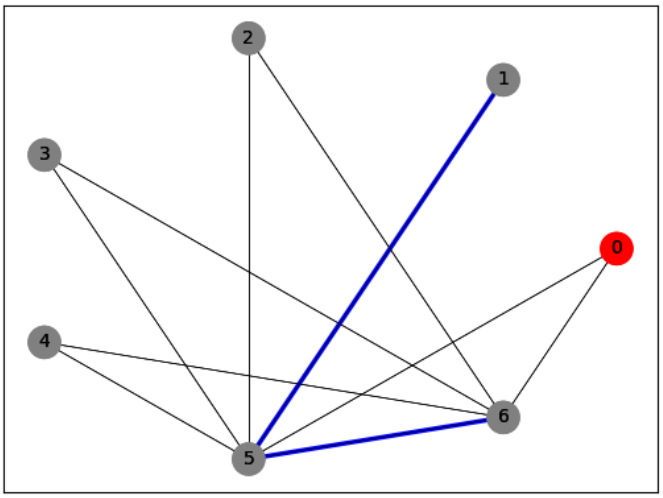
Algorytm 10: *isBridge* rekurencyjna funkcja pomocnicza dla *FleuryAlgorithm*

```
1  def isBridge(adjList, u, v):
2          if len(adjList[u]) == 1:
3                  return True
4          else:
5                  listOfVisitedNeighbours = [False] * len(adjList)
6                  cntBeforeRestoreEdge = DFS(adjList, u, listOfVisitedNeighbours)
7
8                  removeEdge(adjList, u, v)
9                  listOfVisitedNeighbours = [False] * len(adjList)
10                 cntAfterRestoreEdge = DFS(adjList, u, listOfVisitedNeighbours)
11
12                 # restore edge
13                 addEdge(adjList, u, v)
14
15                 return cntBeforeRestoreEdge < cntAfterRestoreEdge
```

3.3.1 Wizualizacja grafu nieskierowanego

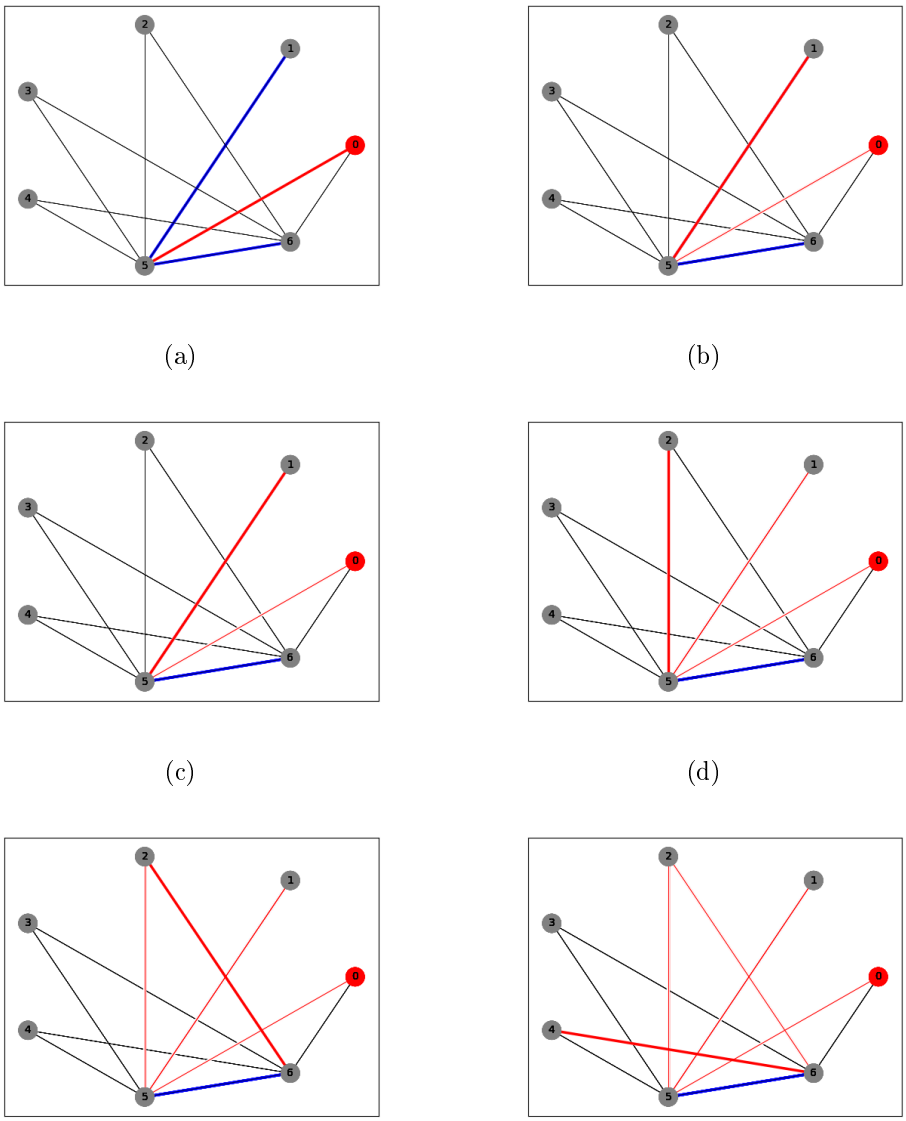


(a)

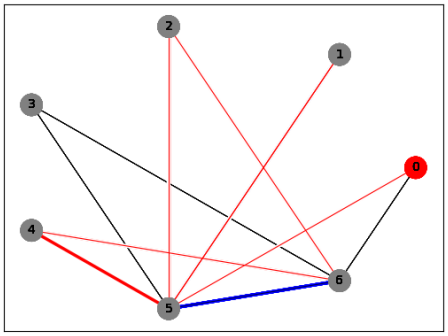


(b)

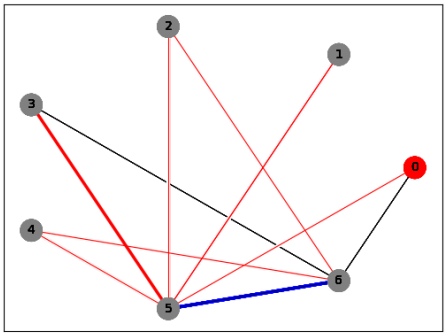
Rysunek 6: Sieć złożona o modelu Barabási–Albert (rozdział 1.5.1), dla parametrów $m_0 = 5, t = 7$ na rysunku 6a. Ta sama sieć po modyfikacji do grafu Eulera na rysunku 6b, na niebiesko zaznaczono krawędzie wielokrotne, stworzone sztucznie algorytmem 1, czerwony wierzchołek jest punktem startowym wyznaczania cyklu.



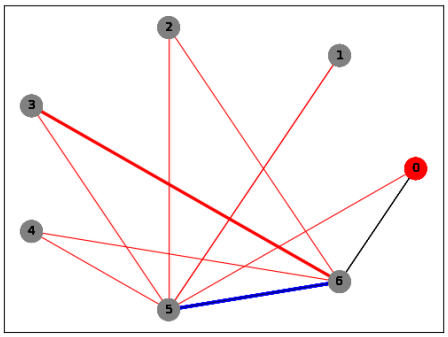
Rysunek 7: Kroki 1-6 algorytmu Fleury'ego na nieskierowanym grafie wejściowym z rysunku 6. Grubą linią czerwoną oznaczono aktualną krawędź, cieńszą krawędzie już odwiedzone, a na niebiesko krawędzie wielokrotne.
Cały cykl Eulera przebiega kolejno przez wierzchołki: 0, 5, 1, 5, 2, 6, 4, 5, 3, 6, 5, 6, 0.



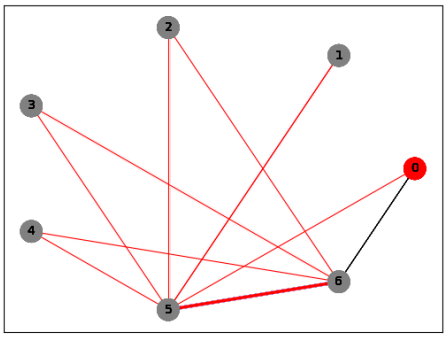
(a)



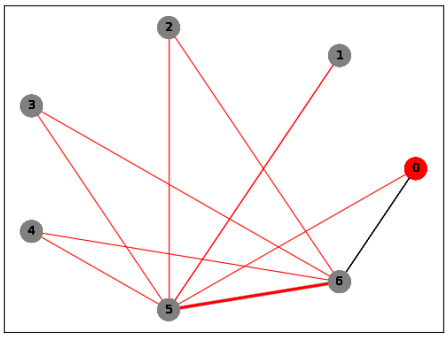
(b)



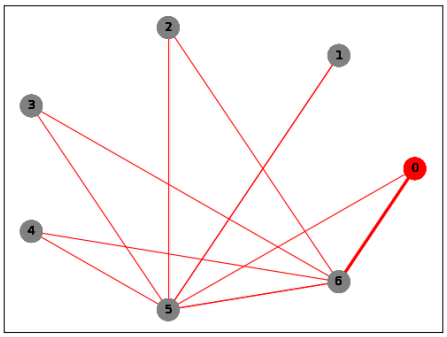
(c)



(d)



(e)



(f)

Rysunek 8: Kontynuacja rysunku 7.

Kroki 7-12 algorytmu Fleury'ego na nieskierowanym grafie wejściowym z rysunku 6. Grubą linią czerwoną oznaczono aktualną krawędź, cieńszą krawędzie już odwiedzone, a na niebiesko krawędzie wielokrotne.

Cały cykl Eulera przebiega kolejno przez wierzchołki: 0, 5, 1, 5, 2, 6, 4, 5, 3, 6, 5, 6, 0.

3.4 Algorytm Hierholzera

Kolejnym zaimplementowanym sposobem znajdowania cyklu Eulera w UG i DG jest algorytm Hierholzera[21]. Tak jak dla algorytmu opisanego w 3.3 wymagane jest, żeby graf był eulerowski, co w tym przypadku mogą zapewnić algorytmy 1 i 3, w zależności od wartości zmiennej *isDirected*.

Podobnie jak algorytm z rozdziału 3.3, operuje on na liście sąsiedztwa, co było najbardziej efektywnym wyborem zważywszy na typy danych testowanych (sieci opisane w 1.5). Wierzchołek startowy znany jest z góry. Poszukiwana od niego zaczęte polegają na znajdowaniu mniejszych cykli (domkniętych szlaków) usuwając przy tym przebyte krawędzie. Wykorzystuje się przy tym stos, na który w odpowiedniej kolejności odkłada się odwiedzone wierzchołki.

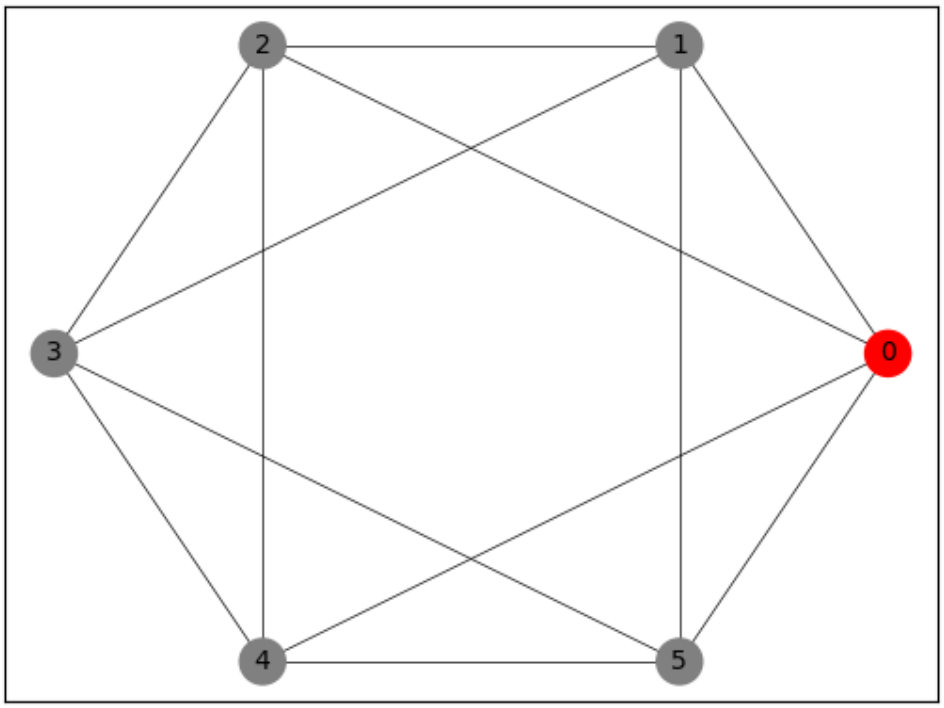
Algorytm 11: *HierholzerAlgorithm* funkcja wyznaczająca ścieżkę dla grafów skierowanych i nieskierowanych

```
1  def HierholzerAlgorithm(graph, isDirected, startNode):
2          adjList = GraphHelper.getAdjList(graph)
3          EulerCycle = list()
4          stackOfNodes = deque()
5
6          node = startNode
7          EulerCycle.append(node)
8
9          while True:
10                 if checkOutDegree(adjList, node):
11                         v = getNextOutEdge(adjList, node)
12                         stackOfNodes.append(node)
13                         removeEdge(adjList, node, v, isDirected)
14                         node = v
15                 else:
16                         node = stackOfNodes.pop()
17                         EulerCycle.append(node)
18
19                 if not stackOfNodes:
20                         break
21
22         EulerCycle.reverse()
23         return EulerCycle
```

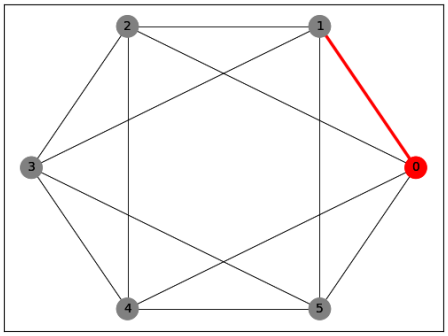
Złożoność obliczeniowa szacowana jest na $O(E)$.

3.4.1 Wizualizacja grafu nieskierowanego

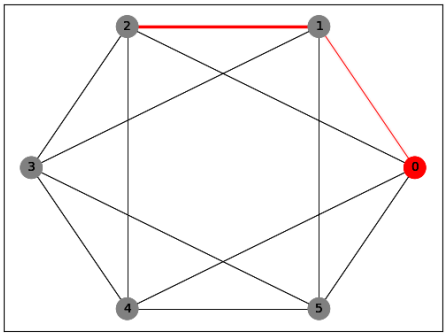
Przebieg algorytmu dla grafu nieskierowanego z rysunku 9, który został stworzony według modelu sieci *small-world*, dla parametrów $n = 6, k = 4, p = 0$, zilustrowano na rysunkach 10 i 11.



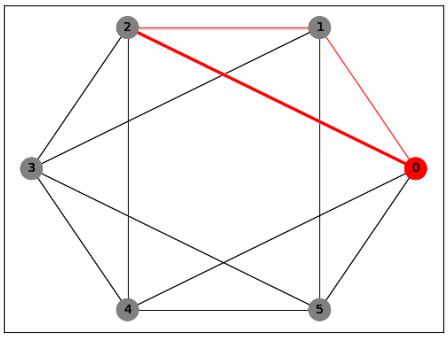
Rysunek 9: Sieć złożona o modelu Wattsa–Strogatza (rozdział 1.5.2), dla parametrów $n = 6, k = 4, p = 0$. Wierzchołek początkowy zaznaczono kolorem czerwonym.



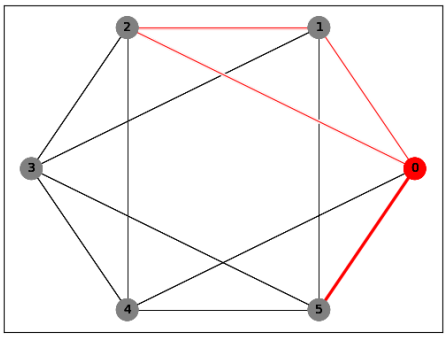
(a)



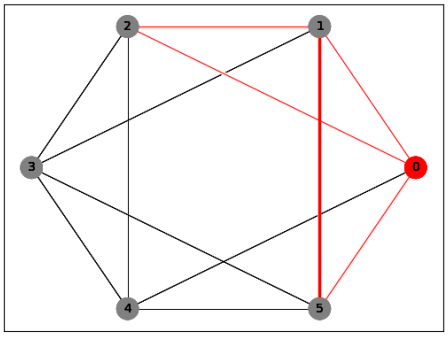
(b)



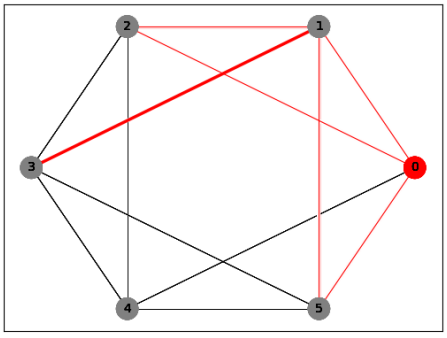
(c)



(d)



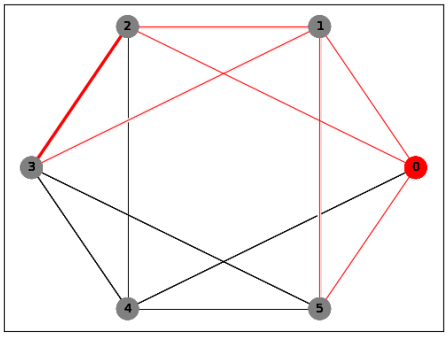
(e)



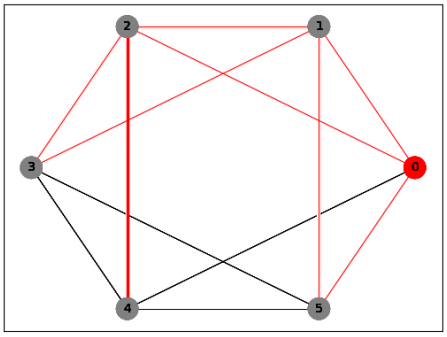
(f)

Rysunek 10: Kroki 1-6 algorytmu Hierholzera na nieskierowanym grafie wejściowym z rysunku 9. Grubą linią czerwoną oznaczono aktualną krawędź, cieńszą krawędzie już odwiedzone.

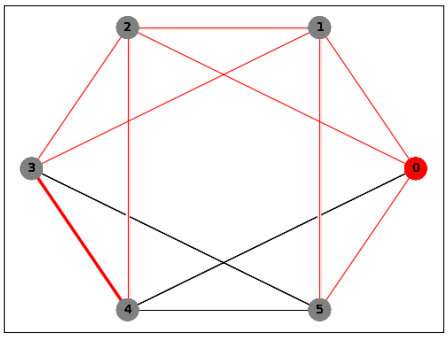
Cały cykl Eulera przebiega kolejno przez wierzchołki: 0, 1, 2, 0 , 5, 1, 3, 2, 4, 3, 5, 4, 0.



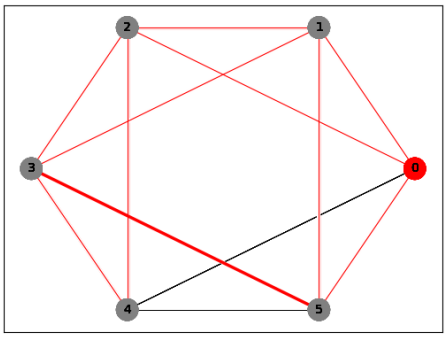
(a)



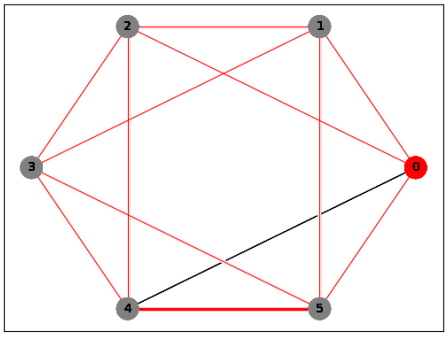
(b)



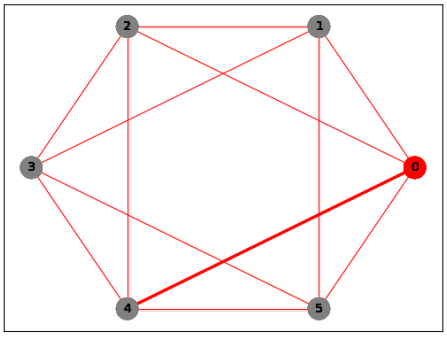
(c)



(d)



(e)

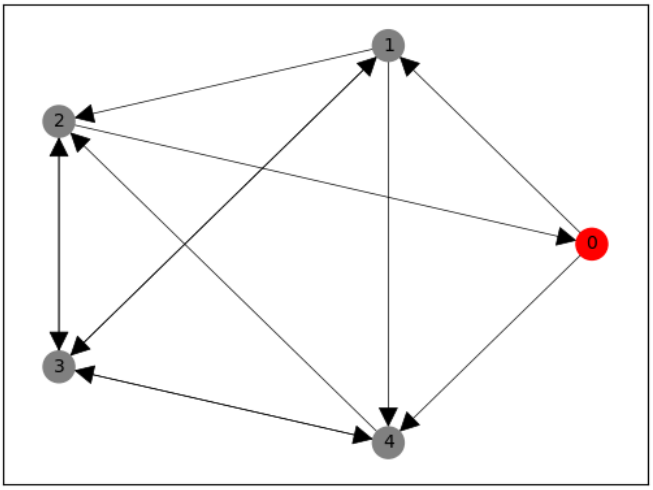


(f)

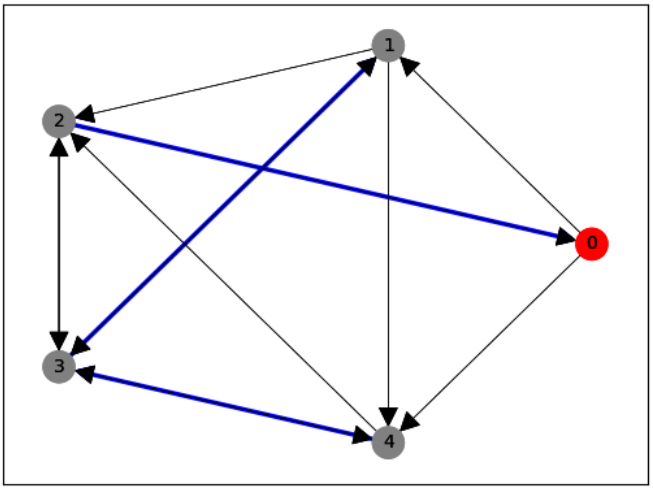
Rysunek 11: Kontynuacja rysunku 10.

Kroki 7-12 algorytmu Hierholzera na nieskierowanym grafie wejściowym z rysunku 9. Grubą linią czerwoną oznaczono aktualną krawędź, cieńszą krawędzie już odwiedzone. Cały cykl Eulera przebiega kolejno przez wierzchołki: 0, 1, 2, 0 , 5, 1, 3, 2, 4, 3, 5, 4, 0.

3.4.2 Wizualizacja grafu skierowanego

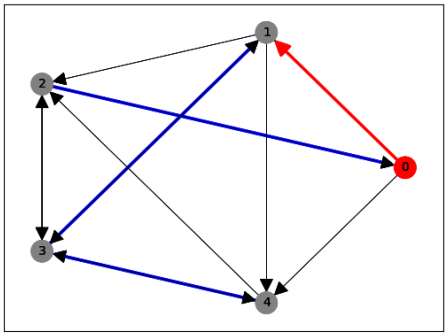


(a)

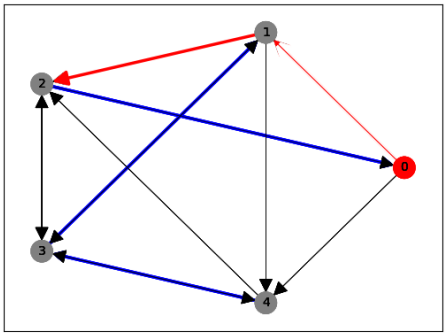


(b)

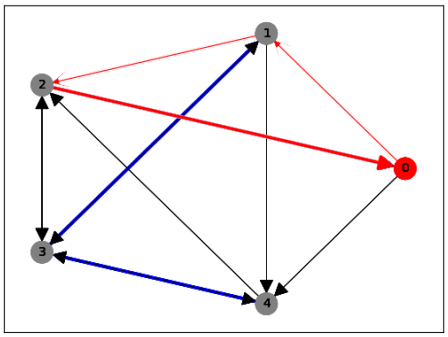
Rysunek 12: Losowy graf skierowany o parametrach $n = 5, p = 0.5$, gdzie n to ilość wierzchołków, a p to prawdopodobieństwo utworzenia krawędzi na rysunku 12a. Ten sam graf po modyfikacji do grafu Eulera na rysunku 12b, na niebiesko zaznaczono krawędzie wielokrotne, stworzone sztucznie algorytmem nr 3, czerwony wierzchołek jest punktem startowym wyznaczania cyklu.



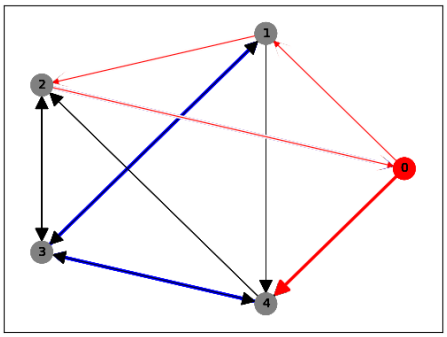
(a)



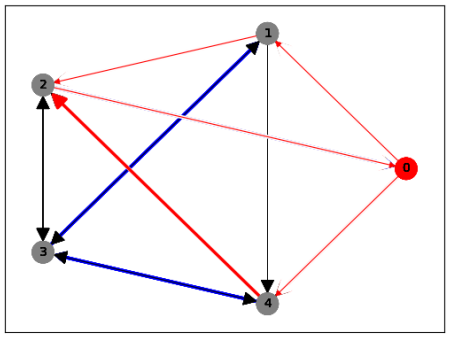
(b)



(c)

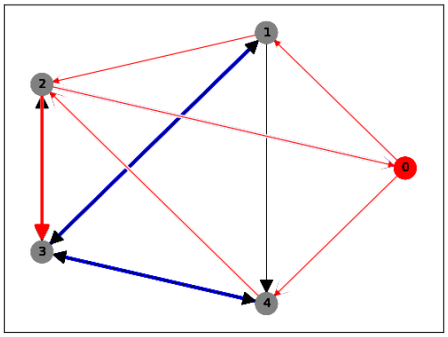


(d)

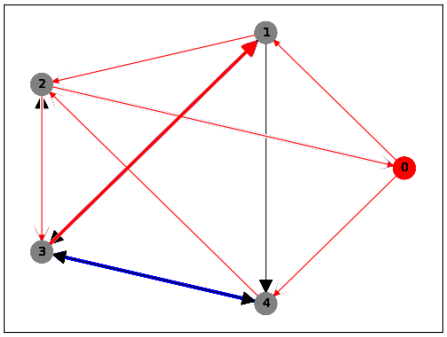


(e)

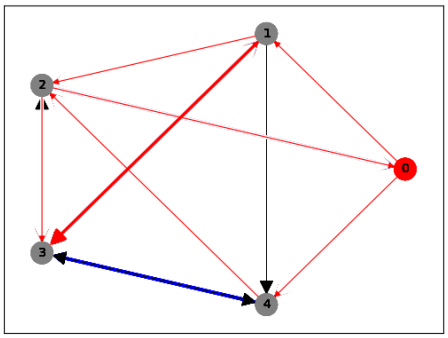
Rysunek 13: Kroki 1-5 algorytmu Hierholzera na skierowanym grafie wejściowym z rysunku 12. Grubą linią czerwoną oznaczono aktualną krawędź, cieńszą krawędzie już odwiedzone, a na niebiesko krawędzie wielokrotne.
Cały cykl Eulera przebiega kolejno przez wierzchołki: 0, 1, 2, 0, 4, 2, 3, 1, 3, 1, 4, 3, 4, 3, 2, 0.



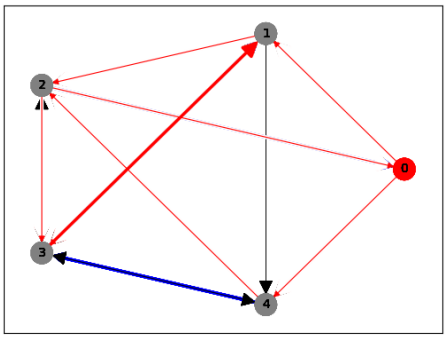
(a)



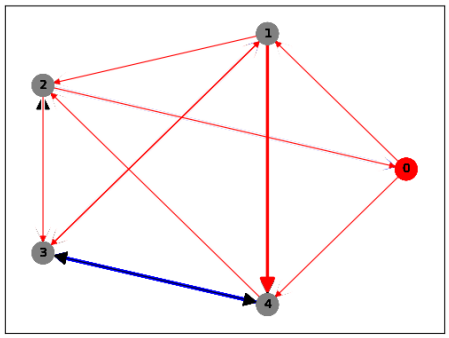
(b)



(c)

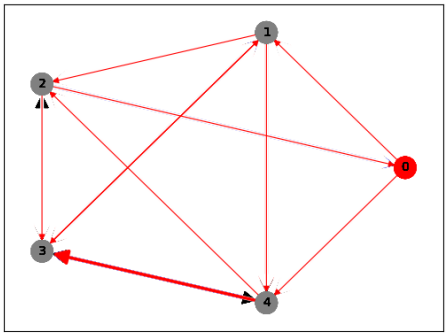


(d)

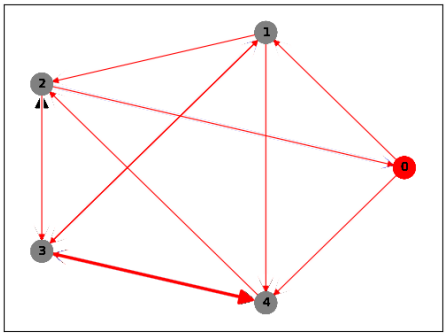


(e)

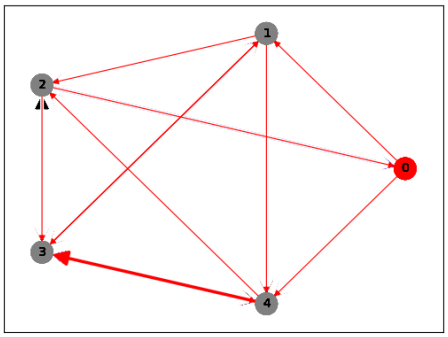
Rysunek 14: Kontynuacja rysunku 13.
Kroki 6-10 algorytmu Hierholzera na skierowanym grafie wejściowym z rysunku 12. Grubą linią czerwoną oznaczono aktualną krawędź, cieńszą krawędzie już odwiedzone, a na niebiesko krawędzie wielokrotne.
Cały cykl Eulera przebiega kolejno przez wierzchołki: 0, 1, 2, 0, 4, 2, 3, 1, 3, 1, 4, 3, 4, 3, 2, 0.



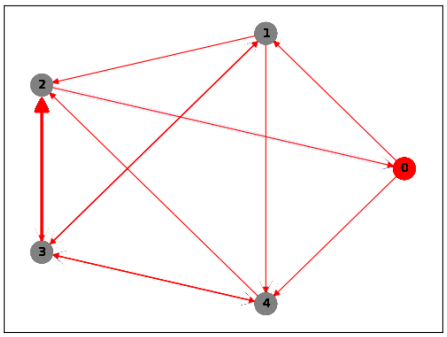
(a)



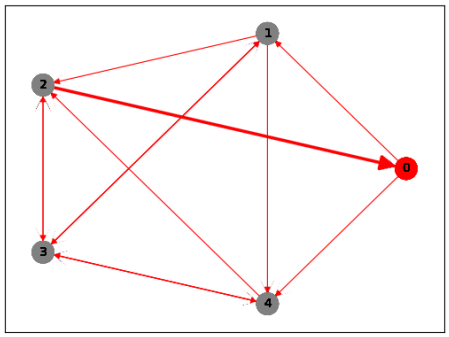
(b)



(c)



(d)



(e)

Rysunek 15: Kontynuacja rysunku 14.
Kroki 11-15 algorytmu Hierholzera na skierowanym grafie wejściowym z rysunku 12. Grubą linią czerwoną oznaczono aktualną krawędź, cieńszą krawędzie już odwiedzone. Cały cykl Eulera przebiega kolejno przez wierzchołki: 0, 1, 2, 0, 4, 2, 3, 1, 3, 1, 4, 3, 4, 3, 2, 0.

4 Wyniki

4.1 Generowanie grafów losowych

Podczas generowania sieci złożonych Barabási–Alberta i Wattsa–Strogatza, które są szerzej opisane w rozdziale 1.5, zebrano wyniki zaprezentowane odpowiednio w tabelach 1 i 2. Parametry, dla których tworzono sieci o modelu BA to $m_0 = w$, $t = \frac{w}{2}$, gdzie w jest liczbą wierzchołków. Przy generowaniu losowych sieci WS, wybrano następujące wielkości: $n = w$, $k = \frac{w}{2}$, $p = 0.75$, gdzie w - liczba węzłów.

Tabela 1: Wyniki generowania sieci złożonych o modelu Barabási–Alberta, średnie wartości obliczono z 5 próbek testowych.

ilość wierzchołków	śr. ilość krawędzi	śr. czas generowania grafu [s]
20	107.6	0.1611
50	651.0	0.5011
80	1638.8	2.0973
100	2548.6	4.2081
120	3660.0	7.2537
150	5699.8	14.7364
180	8192.6	26.9951
200	10091.2	34.1516
220	12205.8	59.1929
250	15740.0	87.9036
500	62727.0	674.8682
700	122825.0	2145.4006
1000	250455.2	4536.7447
2000	1000931.0	41671.2410

Porównując wyniki z tabel 1 i 2 łatwo można zauważyć, że pomimo podobnej ilość krawędzi w grafach, czasy ich generowania (uwzględniające funkcję

Tabela 2: Wyniki generowania sieci złożonych o modelu Wattsa–Strogatza, średnie wartości obliczono z 5 próbek testowych.

ilość wierzchołków	śr. ilość krawędzi	śr. czas generowania grafu [s]
20	108.6	0.0452
50	624.8	0.6187
80	1639.6	2.6913
100	2549.2	5.7963
120	3656.4	10.1062
150	5628.0	19.9065
180	8184.6	34.0788
200	10098.6	47.2212
220	12204.2	62.1643
250	15618.0	91.3247
500	62737.2	732.4247
700	122820.4	165.9884
1000	250480.6	5070.8936
2000	1000950.8	47148.0105

z rozdziału 3.1.1) zaczynają się rozbiegać już w okolicach 100 wierzchołków, na korzyść modelu Barabási–Alberta.

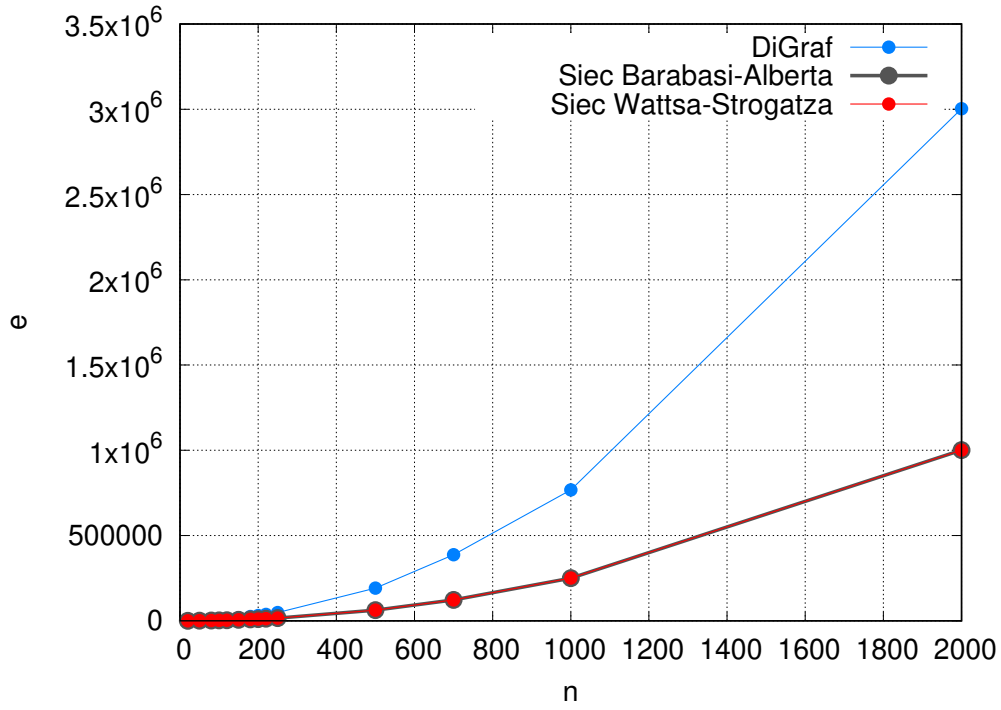
Wyniki z tabeli 3 znacznie różnią się od 1 i 2, spowodowane jest to innym typem wejściowym (w tym przypadku jest to DG), a co za tym idzie, funkcja modyfikująca do grafu Eulera jest inna. Warunki twierdzenia o grafie skierowanym (tw. 2) wpływają na ilość obliczeń możliwych permutacji.

Do generowania skierowanych grafów losowych, których dane zebrano w tabeli 3, wykorzystano funkcję *fast_gnp_random_graph* z biblioteki *Networkx*, która przyjmuje wartości n i p, gdzie n - ilość wierzchołków, p - prawdopodobieństwo stworzenia krawędzi (w przypadku stworzonych danych obrano jego stałą wartość: $p = 0.75$).

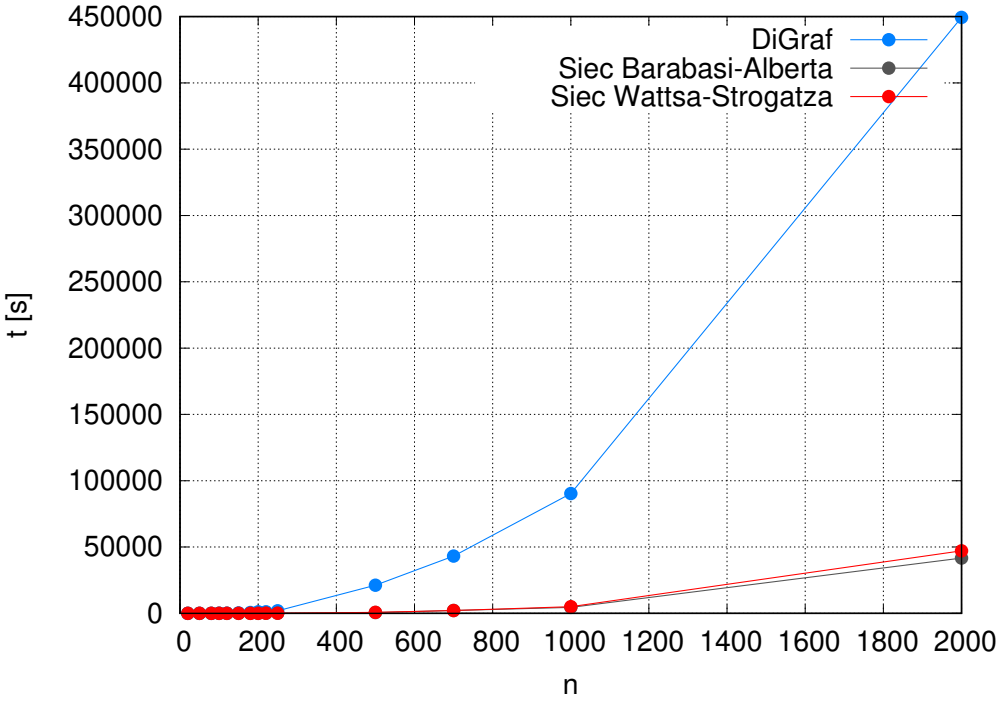
Tabela 3: Wyniki generowania grafu losowego, średnie wartości obliczono z 5 próbek testowych.

ilość wierzchołków	śr. ilość krawędzi	śr. czas generowania grafu [s]
20	323.2	0.1844
50	2032.8	5.5678
80	5144.0	43.1782
100	7892.2	76.6788
120	11297.6	136.4873
150	17611.2	306.0416
180	25294.8	599.1041
200	31180.2	1143.3872
220	37617.4	1213.5494
250	48416.0	1918.1350
500	192001.6	21215.9172
700	387793.2	43215.4681
1000	767803.0	90381.0973
2000	3003152.6	219311.4794

Wyniki zgromadzone w tabelach 1, 2 i 3 przeniesione zostały na wykresy z rysunków 16a, 16b. Na pierwszym z nich można zauważyć znaczą różnicę w ilości krawędzi pomiędzy grafem skierowanym a sieciami złożonymi, natomiast te drugie (modele WS i BA) są sobie bardzo bliskie, ich funkcję nakładają się na siebie. Ma to swoje przełożenie na czas generowania takich grafów (rys. 16b), gdzie funkcja DG rośnie bardzo szybko, z znaczną różnicą względem dwóch pozostałych. Natomiast dla sieci Barabási–Alberta i Watsa–Strogatza czasy zaczynają się zauważalnie różnić powyżej 1000 wierzchołków.



(a) Zależność stworzonych krawędzi dla każdego typu grafów testowych w zależności od ilości wierzchołków, gdzie n - liczba wierzchołków, e - liczba krawędzi.



(b) Czas generowania grafów, uwzględniając funkcję modyfikującą do grafu Eulera, gdzie n - liczba węzłów, t - czas podany w sekundach.

Rysunek 16: Wykresy porównujące dane z tabel 1, 2 i 3.

4.2 Algorytm Fleury'ego

Wyniki z znajdowania cyklu Eulera dla grafów nieskierowanych przy pomocy algorytmu Fleury'ego zostały zaprezentowane w tabelach 4 i 5.

Stos w języku Python posiada limit zapobiegający nieskończonej rekurencji i zbyt wielkiemu wykorzystaniu zasobów pamięciowych, co mogłoby skutkować awarią systemu. Ponieważ funkcja ta działa rekurencyjnie, nawet przy zwiększeniu limitu do maksymalnej wartości, wykonano pomiary tylko do 280 węzłów.

Tabela 4: Czasy wyszukiwania cyklu Eulera sieci złożonych o modelu Barabási–Alberta, dla algorytmu Fleury'ego, średnie wartości obliczono z 5 próbek testowych.

ilość wierzchołków	śr. ilość krawędzi	śr. czas szukania ścieżki
20	107.6	0.0025
40	418.4	0.0331
50	651.0	0.0496
60	930.2	0.0941
70	1258.8	0.1438
80	1638.8	0.2552
90	2072.4	0.4181
100	2548.6	0.5565
120	3660.0	1.1322
150	5699.8	2.7246
180	8192.6	5.4286
200	10091.2	8.1718
220	12205.8	11.8271
250	15740.0	19.0686
280	19730.4	30.2086

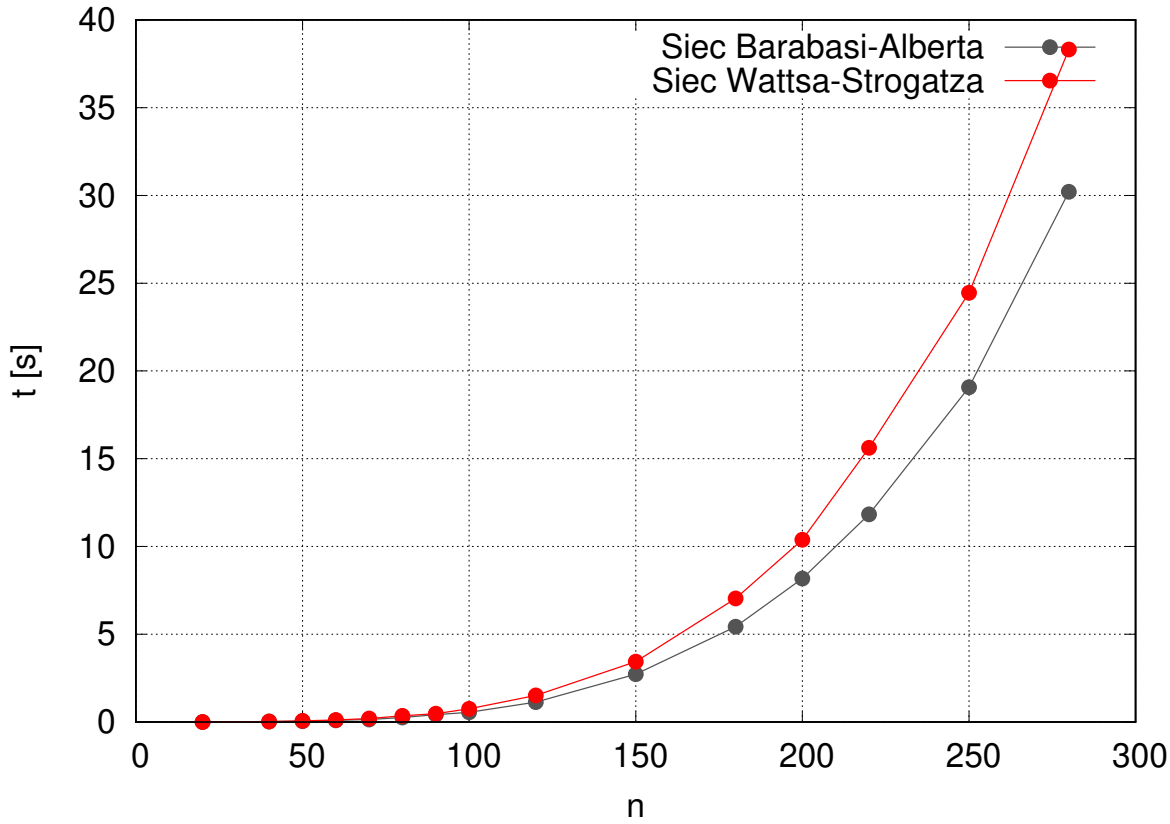
Analizując rysunek 17, na którym przedstawiono wykresy powstałe z danych z dwóch tabel reprezentujących czasy wykonania algorytmu dla modeli sieci złożonych Barabási–Alberta i Wattsa–Strogatza (tab. 4, 5), możliwe jest zauważenie, że dla początkowych mniejszych ilości wierzchołków nie ma większej różnicy. Wprowadzając oznaczenie $\Delta t(n) = t_{WattsaStrogatz}(n) - t_{BarabsiAlbert}(n)$, można określić kilka różnic.

Tabela 5: Czasy wyszukiwania cyklu Eulera sieci złożonych o modelu Wattsa–Strogatza, dla algorytmu Fleury'ego, średnie wartości obliczono z 5 próbek testowych.

ilość wierzchołków	śr. ilość krawędzi	śr. czas szukania ścieżki
20	107.6	0.0038
40	422.4	0.0265
50	622.2	0.0653
60	928.0	0.1114
70	1225.6	0.2043
80	1641.2	0.3526
90	2021.4	0.4789
100	2547.0	0.7555
120	3656.2	1.5154
150	5625.8	3.4387
180	8192.6	7.0422
200	10099.2	10.3851
220	12203.6	15.6236
250	15628.8	24.4495
280	19733.4	38.3141

Zgodnie z danymi z tabel wyznaczono różnice dla trzech ilości n $\Delta t(100) \approx 0.2s$, $\Delta t(200) \approx 2.2s$ i $\Delta t(280) \approx 8.1s$. Czasy te nie są mocno rozbieżne,

ale należy mieć na uwadze, że ilość węzłów jest niestety mała i ograniczona przez wymagania sprzętowe. Dla większej ilość wierzchołków różnica może być drastyczna.



Rysunek 17: Czas obliczania ścieżki w grafach wykorzystujący algorytm Fleury'ego, gdzie n - liczba wierzchołków, t - czas podany w sekundach.

4.3 Algorytm Hierholzera

Wyniki drugiego z zaimplementowanych algorytmów do znajdowania cyklu Eulera dla grafów skierowanych i nieskierowanych przedstawiono w tabelach 6, 7 i 8.

Tabela 6: Wyniki testowania sieci złożonych o modelu Barabási–Alberta, dla algorytmu Hierholzera, średnie wartości obliczono z 5 próbek testowych.

ilość wierzchołków	śr. ilość krawędzi	śr. czas szukania ścieżki
20	107.6	0.0005
50	651.0	0.0031
80	1638.8	0.0077
100	2548.6	0.0179
120	3660.0	0.0215
150	5699.8	0.0320
180	8192.6	0.0482
200	10091.2	0.0554
220	12205.8	0.0710
250	15740.0	0.0804
500	62727.0	0.3441
700	122825.0	0.7875
1000	250455.2	1.1889
2000	1000931.0	5.6991

Analizując tabele dla sieci złożonych oraz rysunek 18 można zauważyć, że krzywe reprezentujące te modele, dla mniejszej liczby wierzchołków nachodzą na siebie. Czas wykonania dla zwiększonej liczny n, podobne jak dla poprzedniego algorytmu (rozdział 4.2), jest lepszy dla sieci Barabási–Alberta. Jednak $\Delta t(n) = t_{WattsaStrogatz}(n) - t_{BarabsiAlbert}(n)$ dla rozwiązania Hierholzera przy maksymalnej zbadanej ilości wierzchołków wynosi $\Delta t(2000) = 1.8$

(uwzględnione są tylko sieci złożone).

Tabela 7: Wyniki testowania sieci złożonych o modelu Wattsa–Strogatza, dla algorytmu Hierholzera, średnie wartości obliczono z 5 próbek testowych.

ilość wierzchołków	śr. ilość krawędzi	śr. czas szukania ścieżki
20	108.6	0.0003
50	624.8	0.0033
80	1639.6	0.0105
100	2549.2	0.0141
120	3656.4	0.0208
150	5628.0	0.0383
180	8184.6	0.0495
200	10098.6	0.0591
220	12204.2	0.0766
250	15618.0	0.0865
500	62737.2	0.3775
700	122820.4	0.8825
1000	250480.6	1.4426
2000	1000950.8	7.5426

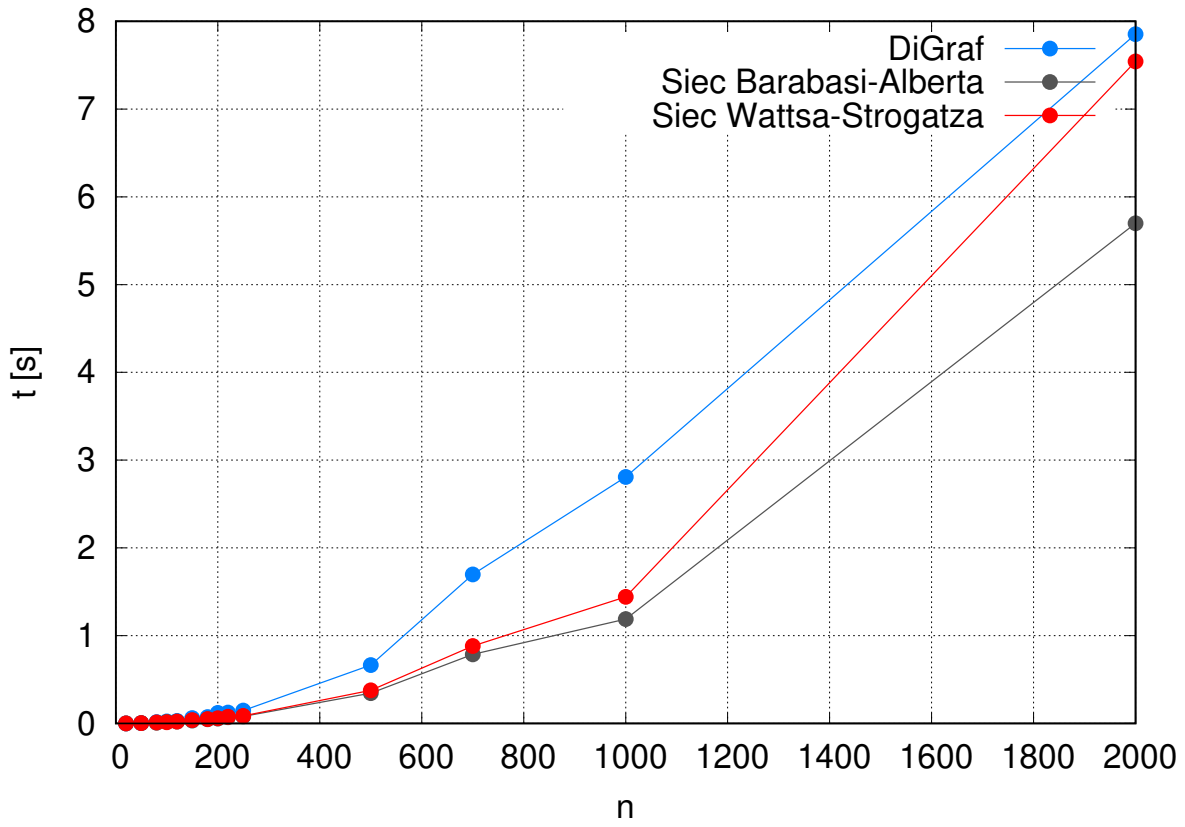
Poza wcześniej wspomnianą informacją, uwagę może zwrócić fakt, że DG występujący w tej analizie, mimo tego, że jest najwolniejszy, to jego czas wykonywania nie różni się drastycznie od wyników dla UG.

Warto wziąć pod uwagę ilość wygenerowanych krawędzi: średnia ilość krawędzi w grafie skierowanym tabela 3, dla grafów nieskierowanych 1 i 2. Różnica pomiędzy krzywymi oznaczonymi na szaro i czerwono, a tą koloru niebieskiego na rys. 18 nie jest duża, a ilość krawędzi w losowym digrafie jest trzykrotnie większa od liczby połączeń w obydwu sieciach złożonych.

Tabela 8: Wyniki testowania grafu skierowanego, dla algorytmu Hierholzera, średnie wartości obliczono z 5 próbek testowych.

ilość wierzchołków	śr. ilość krawędzi	śr. czas szukania ścieżki
20	323.2	0.0006
50	2032.8	0.0042
80	5144.0	0.0146
100	7892.2	0.0229
120	11297.6	0.0289
150	17611.2	0.0613
180	25294.8	0.0733
200	31180.2	0.1209
220	37617.4	0.1245
250	48416.0	0.1476
500	192001.6	0.6641
700	387793.2	1.6977
1000	767803.0	2.8085
2000	3003152.6	7.8537

Czasy do około 200 wierzchołków są bardzo podobne dla każdego badanego typu grafu.



Rysunek 18: Czas obliczania ścieżki w grafach wykorzystujący algorytm Hierholzera, gdzie n - liczba wierzchołków, t - czas podany w sekundach.

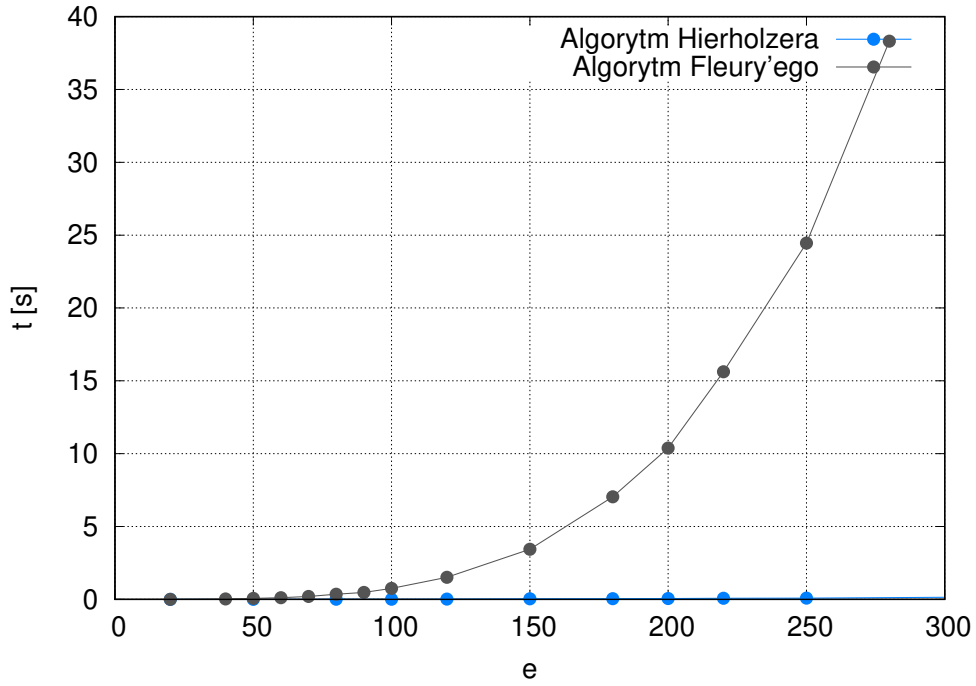
4.4 Algorytm Fleury'ego vs. Hierholzera

Algorytmy porównano względem grafów nieskierowanych, ponieważ jeden z nich działa tylko dla takich.

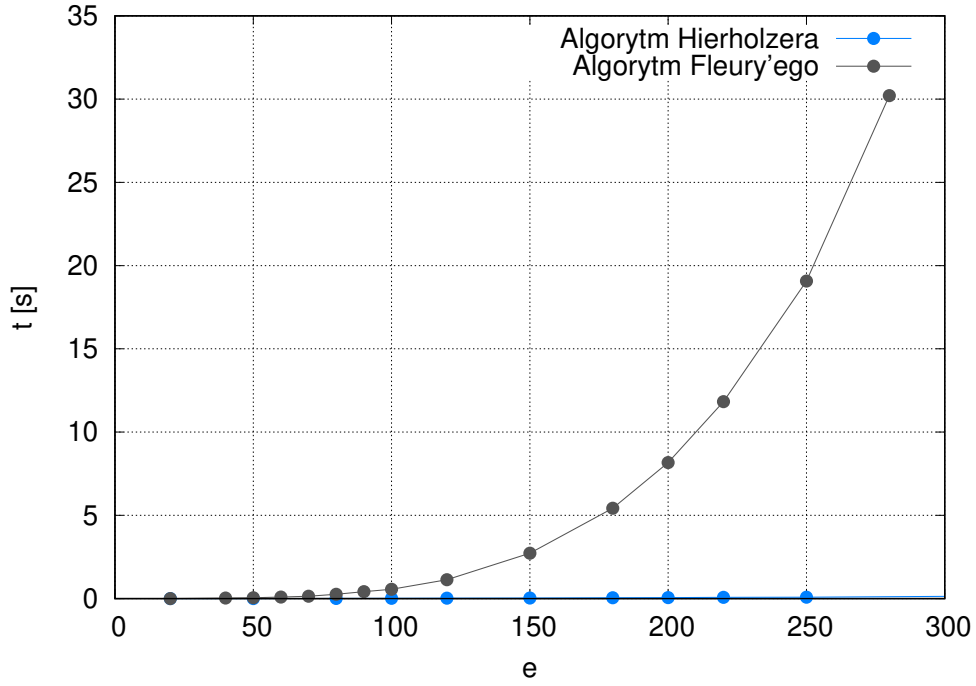
Patrząc na wykresy z rysunku 19, bez większej analizy jesteśmy w stanie określić bardziej optymalne rozwiązanie w szukaniu cyklu Eulera, zarówno dla modelu Wattsa–Strogatza i Barabási–Alberta, którym jest algorytm Hierholzera.

Należy wziąć pod uwagę, że przedstawiono tu porównanie tylko do 280 wierzchołków, ze względu na ograniczenia ilości rekurencji dla algorytmu Fleury'ego, a już widoczna jest znaczna przewaga drugiego z nich.

Maksymalny czas obliczony dla funkcji Hierholzera przy 2000 wierzchoł-



(a) Sieci o modelu Wattsa–Strogatza.



(b) Sieci o modelu Barabási–Alberta.

Rysunek 19: Porównanie czasów obliczania ścieżki w sieciach złożonych algorytmów Hierholzera oraz Fleury'ego, gdzie n - liczba wierzchołków, t - czas podany w sekundach.

kach wynosił $t_{max} \approx 7.5$, dla trudniejszego do przeszukiwań modelu sieci Barabási–Alberta. Dla funkcji Fleury'ego czas przybliżony do t_{max} osiągany jest już pomiędzy 180 węzłami sprawdzanymi, a 200.

4.5 Graf rzeczywisty

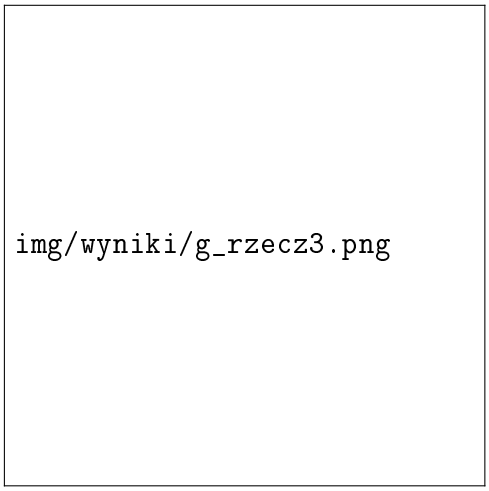
Dla grafu rzeczywistego z rysunku 5 wygenerowano cykle Eulera, przy pomocy funkcji Hierholzera. Stworzony graf miał parametry $n = 211$, $e = 852$, gdzie n to liczba wierzchołków, a e krawędzi.

Czas trwania algorytmu wynosił w przybliżeniu $t \approx 0.0018$. Porównując wynik z tabelą 8 można zauważyć, że dla podobnej ilości wierzchołków różnica ta jest rzędu 10^{-2}, jednak złożoność tego algorytmu to $O(E)$, co oscyluje pomiędzy pierwszy, a drugim wierszem tabeli i zgodne jest z danymi testowymi.

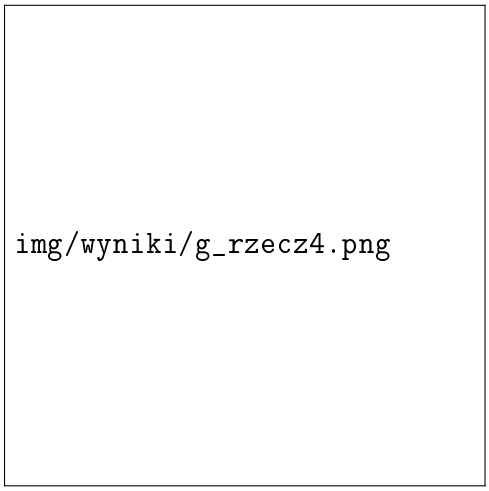
Zaprezentowano je na rysunku 20

(a)

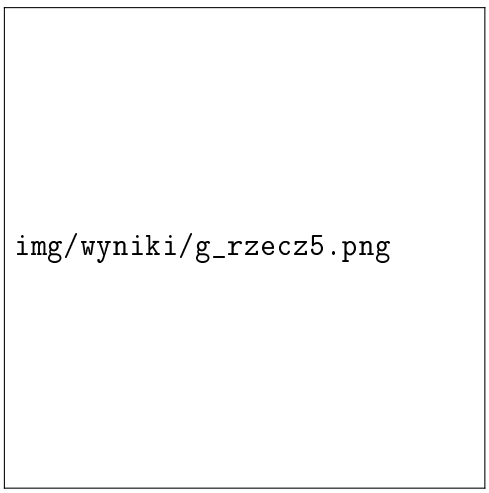
(b)



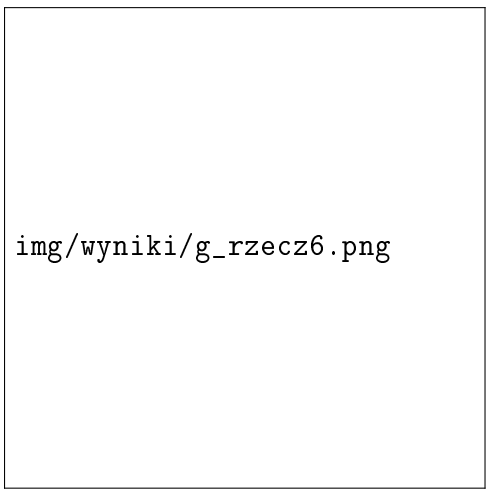
(c)



(d)



(e)



(f)

Rysunek 20: Przebieg algorytmu Hierholzera na grafie powstałym z mapy rzeczywistej z rysunku 4, zaprezentowano niektóre z kroków w cyklu Eulera.

5 Podsumowanie

Dwa programy rozwiązujące problem chińskiego listonosza obrane za cel pracy zostały zaimplementowane. Ich poprawność zilustrowano autorskim programem zaznaczającym przebieg szukanego cyklu Eulera oraz podsumowaniem testów rożnych modeli grafów, gdzie wyniki wyszły zgodne z estymowaną złożonością obliczeniową. Dzięki kilku typom danych testujących (sieci Barabási–Alberta i Wattsa–Strogatza oraz losowy graf skierowany) możliwe było szersze porównanie działania algorytmów, w zależności od typów połączeń, ilości wierzchołków, czy krawędzi w grafie. Ponadto udało się stworzyć graf z danych rzeczywistych, na którym wywołano jeden z napisanych algorytmów.

Funkcja konwertująca graf na eulerowski została zaimplementowana i pomimo braku wszystkich możliwych permutacji, określa najlepsze rozwiązanie na danym zbiorze, które jest wystarczające na potrzeby działań programów.

Algorytm Fleury'ego był znacznie prostszy do stworzenia, ale słabszy wydajnościowo. Problematyczny może być również w jego przypadku limit rekurencji uniemożliwiający zebranie większej ilości danych testowych na przeciętnej klasy sprzęcie. Funkcja Hierholzera rozwiązuje problemy zarówno dla grafów skierowany i nieskierowanych, jest znacznie szybsza, co całkowicie przeważa na jej korzyść.

Innym ciekawym rozszerzeniem pracy mogłoby być porównanie czasów oraz ścieżek wygenerowanych dla grafów rzeczywistych z danymi statystycznymi otrzymanymi po kontakcie z placówką pocztową lub firmą kurierską.

Literatura

[1] M. K. Gordenko, S. M. Avdoshin *The Variants of Chinese Postman Problems and Way of Solving through Transformation into Vehicle Routing Problems.* Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, s. 221-232, 2018

[2] H. A. Eiselt, M. Gendreau, G. Laporte *Arc Routing Problems, Part I: The Chinese Postman Problem* Institute for Operations Research and the Management Sciences (INFORMS), 1995

[3] M. Beck, D. Blado, J. Crawford, T. Jean-Louis, M. Young *On weak chromatic polynomials of mixed graphs*, Graphs and Combinatorics, 2013

[4] R. K. Ahuja, T. L. Magnanti, J. B. Orlin *Network Flows: Theory, algorithmsand applications*, Prentice Hall, New Jersey, s. 740-745, 1993

[5] J. Edmonds, E.L. Johnson *Matching Euler tours and the Chinese postman problem.* Mathematical Programming, s. 88–124, 1973

[6] L. Euler *Solutio problematis ad geometriam situs pertinentis* (ang.), 1741

[7] A.-L. Barabási, R. Albert *Emergence of Scaling in Random Networks*, Science 286, 509-512, 1999

[8] S. Milgram *The Small World Problem* Psychology Today, Ziff-Davis Publishing Company, 1967

[9] J. Watts, S.H. Strogatz *Collective dynamics of 'small-world' networks*, Nature, 393 (6684) s. 440–442, 1998

[10] M. D. Humphries, K. Gurney *Network 'Small-World-Ness': A Quantitative Method for Determining Canonical Network Equivalence*, PLOS ONE, 3 (4), 2008

[11] R. Kasprzyk *Własności sieci z złożonych posiadających cechy Small World i Scale Free* BIULETYN INSTYTUTU SYSTEMÓW INFORMATYCZNYCH 1 25-30, 2008

[12] https://pl.python.org/ [dostęp: 23.12.2020]

[13] https://networkx.org/ [dostęp: 23.12.2020]

[14] https://matplotlib.org/ [dostęp: 23.12.2020]

[15] https://www.mathworks.com/help/matlab/ [dostęp: 23.12.2020]

[16] https://www.openstreetmap.org/ [dostęp: 23.12.2020]

[17] https://wiki.openstreetmap.org/wiki/Category:Properties [dostęp: 20.12.2020]

[18] https://www.w3.org/XML/ [dostęp: 25.12.2020]

[19] https://www.movable-type.co.uk/scripts/latlong.html [dostęp: 25.12.2020]

[20] T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012

[21] Z. Qiu, Z. Liu, X. Zhang *Tweaking for Better Algorithmic Implementation* Computer Science Department, Southeast Missouri State University, Cape Girardeau, MO, U. S. A. CAINE 2017, San Diego, California, USA, 2017