Metody Obliczeniowe w Nauce i Technice

Laboratorium 2

Wojciech Łącki

SPIS TREŚCI

ZADANIE 1 – METODA GAUSSA-JORDANA

Polecenie: Napisz i sprawdź funkcje rozwiązującą układ równań liniowych $n \times n$ metodą Gaussa-Jordana z częściowym poszukiwaniem elementu wiodącego.

```
import random
import numpy as np
import time
import matplotlib.pyplot as plt
import scipy.linalg


def complete(A):
  n = len(A)
  solution = [0 for _ in range(n)]
  T = []

  def findBiggestVal(A, start):
    n = len(A)
    y, x = start, start
    for i in range(start, n):
      for j in range(start, n):
        if abs(A[i][j]) > abs(A[y][x]):
          y, x = i, j
    return y, x

  for i in range(n):
    y, x = findBiggestVal(A, i)
    if x != i:
      T.append((x, i))
    for j in range(n + 1):
      A[i][j], A[y][j] = A[y][j], A[i][j]
    for j in range(n):
      A[j][x], A[j][i] = A[j][i], A[j][x]
    if A[i][i] == 0:
      print("Nie dzielimy przez 0")
      break
    for j in range(i + 1, n):
      factor = A[j][i] / A[i][i]
      for k in range(n + 1):
        A[j][k] -= (A[i][k] * factor)
    for j in range(i - 1, -1, -1):
      factor = A[j][i] / A[i][i]
      for k in range(n + 1):
        A[j][k] -= (A[i][k] * factor)

  for i in range(n):
    solution[i] = A[i][n] / A[i][i]
    A[i][n] = solution[i]
    A[i][i] = 1

  for i in range(len(T) - 1, -1, -1):
```

```
    y, x = T[i]
    A[y][n], A[x][n] = A[x][n], A[y][n]
    solution[y], solution[x] = solution[x], solution[y]

  print(T)
  print(solution)


def partial(A):
  n = len(A)
  solution = [0 for _ in range(n)]

  def findBestRow(A, start, column):
    n = len(A)
    best = start
    for i in range(start + 1, n):
      if abs(A[i][column]) > A[best][column]:
        best = i
    return best

  for i in range(n):
    index = findBestRow(A, i, i)
    for j in range(n + 1):
      A[i][j], A[index][j] = A[index][j], A[i][j]
    if A[i][i] == 0:
      print("Nie dzielimy przez 0")
      break
    for j in range(i + 1, n):
      factor = A[j][i] / A[i][i]
      for k in range(n + 1):
        A[j][k] -= (A[i][k] * factor)
    for j in range(i - 1, -1, -1):
      factor = A[j][i] / A[i][i]
      for k in range(n + 1):
        A[j][k] -= (A[i][k] * factor)

  for i in range(n):
    solution[i] = A[i][n] / A[i][i]
    A[i][n] = solution[i]
    A[i][i] = 1

  return solution


def scaling(A):
  n = len(A)
  solution = [0 for _ in range(n)]
  for i in range(n):
    if A[i][i] == 0:
      print("Nie dzielimy przez 0")
      break
    for j in range(i + 1, n):
```

```
    factor = A[j][i] / A[i][i]
    for k in range(n + 1):
      A[j][k] -= (A[i][k] * factor)
  for j in range(i - 1, -1, -1):
    factor = A[j][i] / A[i][i]
    for k in range(n + 1):
      A[j][k] -= (A[i][k] * factor)

 for i in range(n):
  solution[i] = A[i][n] / A[i][i]
  A[i][n] = solution[i]
  A[i][i] = 1
 return solution
```

Powyżej napisane jest program w trzech wersjach:

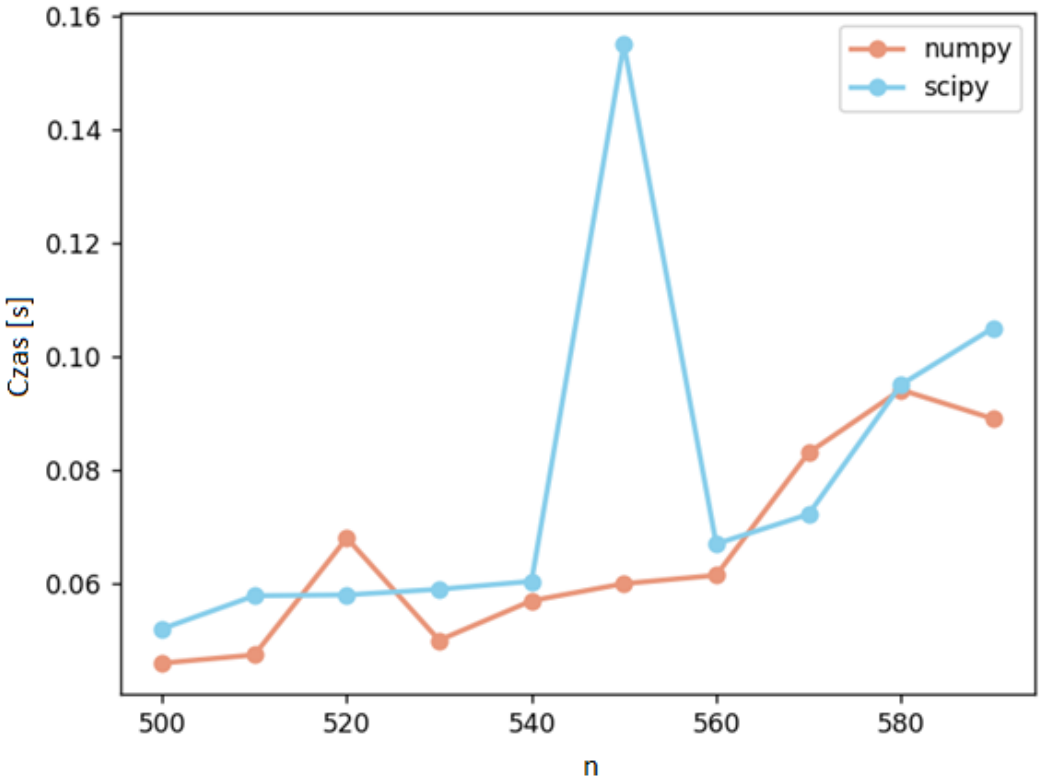
- Complete pivoting – pełne poszukiwanie elementu wiodącego (szukanie w wierszach i kolumnach)
- Partial pivoting – częściowe poszukiwanie elementu wiodącego (szukanie jedynie w kolumnie)
- Scaling – podstawowe skalowanie

Dalsza część polecenia: Dla dziesięciu różnych rozmiarów macierzy współczynników większych niż 500 × 500 porównaj czasy działania zaimplementowanej funkcji z czasami uzyskanymi dla wybranych funkcji bibliotecznych.

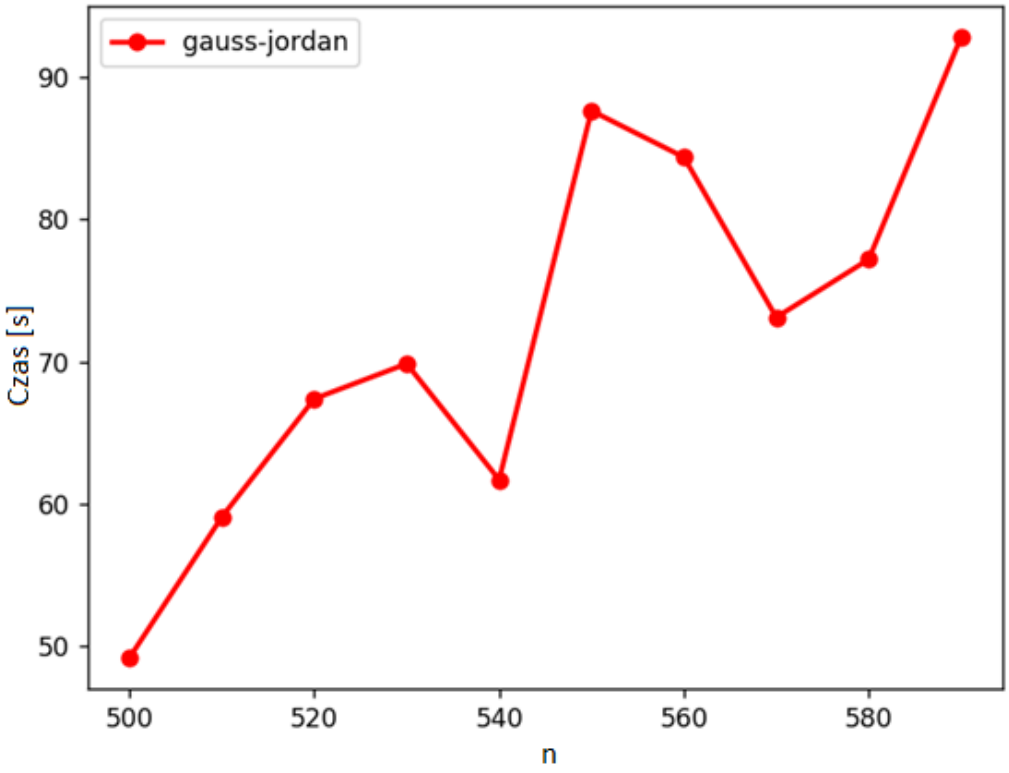
```
def test(n):
 print(n)
 A = [[random.randint(-1000, 1000) for _ in range(n)] for _ in range(n)]
 B = [random.randint(-1000, 1000) for _ in range(n)]

 C = [[0 for _ in range(n + 1)] for _ in range(n)]
 for i in range(n):
  for j in range(n):
   C[i][j] = A[i][j]
 for i in range(n):
  C[i][n] = B[i]

 start = time.time()
 npSolution = np.linalg.solve(A, B)
 end = time.time()
 diff = end - start
 print("Numpy:", diff)
 npTimes.append(diff)

 start = time.time()
 scSolution = scipy.linalg.solve(A, B)
 end = time.time()
 diff = end - start
 print("Scipy:", diff)
 scTimes.append(diff)
```

```
  start = time.time()
  # implementedSolution = scaling(C)
  implementedSolution = partial(C)
  # implementedSolution = complete(C)
  end = time.time()
  diff = end - start
  print("Implemented:", diff)
  implementedTimes.append(diff)
  print("Close numpy scipy: ", np.allclose(npSolution, scSolution))
  print("Close numpy implemented: ", np.allclose(npSolution, implementedSolution))
  print("Close scipy implemented: ", np.allclose(scSolution, implementedSolution))
  matrixSizes.append(n)


npTimes = []
scTimes = []
implementedTimes = []
matrixSizes = []

for n in range(500, 600, 10):
  test(n)

plt.plot(matrixSizes, npTimes, label='numpy', color='darksalmon', marker='o', linewidth=2)
plt.plot(matrixSizes, scTimes, label='scipy', color='skyblue', marker='o', linewidth=2)
plt.legend()
plt.show()
plt.plot(matrixSizes, implementedTimes, label='gauss-jordan', color='red', marker='o', linewidth=2)
plt.legend()
plt.show()
```

W tej części został wykorzystany jedynie „partial pivoting”, gdyż obliczenia dla takiego rzędu wielkości macierzy obliczenia trwają około minuty. W celu zmiany zaimplementowanej metody wystarczy odkomentować wybrany algorytm (zmienna „implementedSolution”).



Wykres 1. Przedstawia czas potrzebny do obliczenia układu równań złożonego z „n” zmiennych przy pomocy funkcji bibliotecznych.



Wykres 2. Przedstawia czas potrzebny do obliczenia układu równań złożonego z „n” zmiennych przy pomocy zaimplementowanej funkcji używającej „partial pivoting”.

Widzimy, że czasy obliczenia tych samych układów równań dla funkcji bibliotecznych jest bardzo podobny i bardzo szybki. Zwiększenie układu równań o 10 zmiennych oznacza nieznaczny spadek szybkości wykonania się algorytmu, na poziomie części setnych czy dziesiętnych sekundy. Natomiast czas do wykonania się zaimplementowanej funkcji liczony jest już w sekundach, a zwiększenie ilości zmiennych wiąże się różnicą w czasie na poziomie kilku sekund.

Tak duża różnica między zaimplementowaną funkcją a tymi bibliotecznymi jest prawdopodobnie związana z dużymi optymalizacjami zastosowanymi w bibliotekach.

Należy jednak zaznaczyć, że wyniki napisanej funkcji są poprawne.

```
Numpy: 0.08897280693054199
Scipy: 0.10496687889099121
Implemented: 92.82800531387329
Close numpy scipy:  True
Close numpy implemented:  True
Close scipy implemented:  True
```

Rysunek 1. Przedstawia wyniki dla układu równań z 590 zmiennymi (n = 590).

Można zauważyć, że zostały porównane rozwiązania każdej możliwej dwójki algorytmów. Wszystkie wyniki były podobne. I tak było dla każdej badanej wartości „n”.

Złożoność przedstawionych funkcji wynosi $O(n^3)$.

Nie rozwiązywałem układów równań dla większej liczby niewiadomych niż 590, gdyż obliczenia byłyby dość długie, a laptop nie jest w najlepszej kondycji 😊.

ZADANIE 2 – FAKTORYZACJA LU

Polecenie: Napisz i przetestuj funkcje dokonująca faktoryzacji $A = LU$ macierzy A (bez poszukiwania elementu wiodącego). Zadbaj o to, żeby implementacja była *in-situ*.

```python
import numpy as np
import copy
import random


def factorization(LU):
  n = len(LU)
  for i in range(n):
    if LU[i][i] == 0:
      print("Nie dzielimy przez 0")
      break
    for j in range(i + 1, n):
      factor = LU[j][i] / LU[i][i]
      for k in range(i + 1, n):
        LU[j][k] -= (LU[i][k] * factor)
      LU[j][i] = factor
```

Powyższa funkcja dokonuje faktoryzacji macierzy LU. Warto zaznaczyć, że algorytm działa w miejscu („in-situ”).

Dalsza część polecenia: Sprawdź poprawność wyniku obliczając $||A - LU||$.

```
def checkCorrectness(A, LU, eps):
  n = len(A)
  for i in range(n):
    for j in range(n):
      total = 0
      for k in range(n):
        if k <= j and k <= i:
          if k == i:
            total += LU[k][j]
          else:
            total += (LU[i][k] * LU[k][j])
      if abs(A[i][j] - total) > eps:
        return False
  return True


def test(n):
  A = [[random.randint(-1000, 1000) for _ in range(n)] for _ in range(n)]
  A = np.array(A, dtype=float)
  LU = copy.deepcopy(A)
  factorization(LU)
  print("SIZE:", n)
  print("CORRECT:", checkCorrectness(A, LU, 1e-5))


test(5)
test(100)
test(200)
```

Funkcja „checkCorrectness” oblicza wartość w kolejnych komórek macierzy A, a później porównuje z macierzą wejściową.

```
SIZE: 5
CORRECT: True
SIZE: 100
CORRECT: True
SIZE: 200
CORRECT: True
```

Rysunek 1. Przedstawia poprawność obliczeń dla wskazanych wielkości macierzy.

Widzimy, że testy pokazały zgodnie, że obliczenia zostały przeprowadzone poprawnie zarówno dla mniejszych jak i większych układów równań.

Mając rozbitą macierz A na dwie macierze trójkątne L (dolną) oraz U (górną) możemy znacznie przyspieszyć obliczenie rozwiązania równania. Wystarczy wtedy rozwiązać poniższy układ równań.

$$\begin{cases} Ly = B & (1) \\ Ux = y & (2) \end{cases}$$

Rozwiązując równanie 1 otrzymamy „y”. Następnie rozwiązujemy równanie 2, gdzie „x” jest naszym oryginalnie szukanym rozwiązaniem.

Ale jak faktoryzacja przyśpiesza obliczenie wyniku?
Jako że L oraz U są trójkątne to rozwiązanie powyższych równań sprowadza się jedynie do podstawienia (nie mamy jednej pętli). Przez co złożoność obliczeniowa z $O(n^3)$ spada do $O(n^2)$.

ZADANIE 3 – ANALIZA OBWODU ELEKTRYCZNEGO – NADOKREŚLONY UKŁAD RÓWNAŃ

Do rozwiązania importujemy poniższe biblioteki.

```
import math
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
from random import randint
```

Biblioteka „networkx” pozwala wygodne wykonywanie operacji na grafach.

a) Polecenie: Program wczytuje z pliku listę krawędzi grafu nieskierowanego ważonego opisującego obwód elektryczny. Wagi krawędzi określają opór fragmentu obwodu między dwoma węzłami. Wierzchołki grafu identyfikowane są przez liczby naturalne.

```
def getData(file_name):
  directed_edges = []
  f = open(file_name, "r")
  for line in f:
    a = line.strip().split(",")
    for i in range(len(a)):
      a[i] = int(a[i])
    directed_edges.append(a)
  f.close()
  return directed_edges
```

Powyższa funkcja wczyta z pliku krawędzie występujące w grafie. Zawartości pliku z krawędziami przybiera postać, gdzie każda linijka odpowiada jednej krawędzi grafu. Linijka składa się z trzech wartości oddzielonych przecinkiem:

- Numer pierwszego wierzchołka (z niego wyjdzie krawędź)
- Numer drugiego wierzchołka (do którego wchodzi krawędź)
- Waga krawędzi (wartość oporu danej krawędzi)

Graf będzie skierowany, aby w łatwiejszy sposób przygotować sobie układ równań.

b) Polecenie: Dodatkowo wczytuje trójkę liczb (*s, t, E*), przy czym para (*s, t*) wskazuje między którymi węzłami sieci przyłożono siłę elektromotoryczna *E*. Opór wewnętrzny SEM można zaniedbać.

```
def amperage_flow(file_name, stE):
  directed_edges = getData(file_name)
  solve_and_test(directed_edges, stE, draw_graph)

def solve_and_test(directed_edges, stE, drawing):
  graph = create_directed_graph(directed_edges, stE)
  A, B, edges = create_system_of_equations(graph, stE)
  solution = np.linalg.solve(A, B)
  solution_graph = create_amperage_flow_graph(solution, edges)
  drawing(solution_graph)
  eps = 0.5
  correct = check_solution(graph, solution_graph, stE, eps)
  print("CORRECT:", correct)
```

Funkcje te są główną częścią rozwiązania zadania. W nich wykonywane są wszystkie potrzebne operacje.

c) Polecenie: Wykorzystując prawa Kirchhoffa (albo metodę potencjałów węzłowych) znajdź natężenia prądu w każdej części obwodu i przedstaw je na rysunku w postaci grafu ważonego z etykietami (wizualizacja grafu wraz z kolorowymi krawędziami pokazującymi wartość natężenia prądu oraz jego kierunek)

```
def create_directed_graph(resistors, stE):
  graph = nx.DiGraph()
  for a, b, resistance in resistors:
    graph.add_edge(a, b, weight=resistance)
  s, t, E = stE
  if (s, t) not in graph.edges() and (t, s) not in graph.edges():
    graph.add_edge(s, t, weight=0)
  return graph


def create_amperage_flow_graph(solution, edges):
  solution_graph = nx.DiGraph()
  for edge, i in edges.items():
    a, b = edge[0], edge[1]
    if solution[i] < 0:
      a, b = edge[1], edge[0]
    solution_graph.add_edge(a, b, weight=round(abs(solution[i]), 2))
  return solution_graph


def draw_graph(graph):
  plt.figure(figsize=(12, 6))
  labels = nx.get_edge_attributes(graph, 'weight')
  edges, weights = zip(*labels.items())
  pos = nx.spring_layout(graph)
  nx.draw(graph, pos, with_labels=True, node_color='r', edgelist=edges, edge_color=weights,
width=2.0,
        edge_cmap=plt.cm.Blues)
  nx.draw_networkx_edge_labels(graph, pos=pos, edge_labels=labels)
  plt.show()
```

```python
def create_system_of_equations(graph, stE):
    s, t, E = stE
    n = graph.number_of_edges()
    A = [[0 for _ in range(n)] for _ in range(n)]
    B = [0 for _ in range(n)]
    # Przyporządkowanie kolejnym krawędziom numerów identyfikacyjnych
    edges = {x: i for i, x in enumerate(graph.edges())}
    equation_row = 0

    # Drugie Prawo Kirchhoffa
    for cycle in nx.cycle_basis(graph.to_undirected()):
        if equation_row >= n:
            break
        for j in range(len(cycle)):
            v1, v2 = cycle[j - 1], cycle[j]
            # Wpisanie oporów elektrycznych do macierzy na podstawie identyfikatorów krawędzi
            if (v1, v2) in edges:
                A[equation_row][edges[v1, v2]] = graph[v1][v2]['weight']
            else:
                A[equation_row][edges[v2, v1]] = -graph[v2][v1]['weight']
            # Sprawdzenie czy siła elektromotoryczne zawarta jest dostarczana przez krawedź
            if (v1, v2) == (s, t):
                B[equation_row] = E
            elif (v2, v1) == (s, t):
                B[equation_row] = -E
        equation_row += 1

    # Pierwsze Prawo Kirchhoffa
    for v1 in graph.nodes():
        if equation_row >= n:
            break
        # Wychodzace krawedzie
        for v2 in graph[v1]:
            A[equation_row][edges[v1, v2]] = 1
        # Wchodzace krawedzie
        for v2, w in graph.in_edges(v1):
            # print(v1, v2, row, edges[v2, v1])
            A[equation_row][edges[v2, v1]] = -1
        equation_row += 1

    return A, B, edges
```

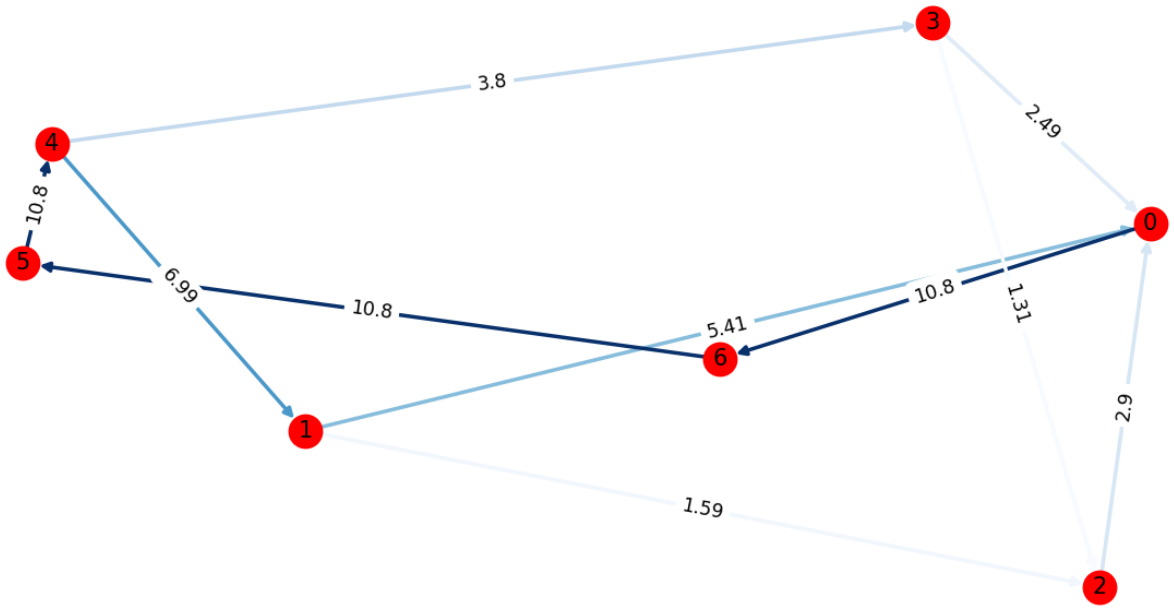
Opis funkcji:

- create_directed_graph – tworzy graf skierowany na podstawie wczytanych z pliku krawędzi oraz informacji, gdzie przyłożony siłę elektromotoryczną.
- create_amperage_flow_graph – tworzy graf skierowany na podstawie rozwiązania układu równań
- draw_graph – rysuje graf przedstawiający rozwiązanie zadania
- create_system_of_equations – tworzy układ równań najpierw na podstawie drugiego prawa Kirchhoffa, a później uzupełnia układ równaniami z pierwszego prawa Kirchhoffa.

Został przeprowadzony test na poprawność czytania zawartości pliku

```
file_name = "graph.txt"
stE = (6, 5, 122)
amperage_flow(file_name, stE)
```

Uzyskaliśmy następujący przepływ prądu w układzie.



Rysunek 1. Przedstawia przepływ prądu w układzie wprowadzonym z pliku.

Po przeanalizowaniu wszystko wygląda w porządku, test sprawdzenia poprawności też dał wynik pozytywny.

```
CORRECT: True
```

Rysunek 2. Przestawia poprawność oblicze przepływ prądu w układzie wprowadzonym z pliku.

d) Polecenie: Przedstaw (wizualizacja + automatyczne sprawdzenie poprawności wyników) działanie programu dla grafów spójnych mających od 15 do 200 wierzchołków.

```python
def check_solution(graph, solution_graph, stE, eps):
 s, t, E = stE
 # Należy przekonwertować na graf nieskierowany, żeby nie było problemów
 graph = graph.to_undirected()
 # Przyporządkowanie kolejnym krawędziom numerów identyfikacyjnych
 edges = {x: i for i, x in enumerate(solution_graph.edges())}

 # Drugie Prawo Kirchhoffa
 for cycle in nx.cycle_basis(graph):
  voltage = 0
  for j in range(len(cycle)):
   v1, v2 = cycle[j - 1], cycle[j]
   if (v1, v2) in edges:
    voltage -= solution_graph[v1][v2]["weight"] * graph[v1][v2]["weight"]
   else:
    voltage += solution_graph[v2][v1]["weight"] * graph[v1][v2]["weight"]
   # Sprawdzenie czy siła elektromotoryczne zawarta jest dostarczana przez krawedź
   if (v1, v2) == (s, t):
    voltage += E
   elif (v2, v1) == (s, t):
    voltage -= E
  if abs(voltage) > eps:
   return False

 # Pierwsze Prawo Kirchhoffa
 for v1 in solution_graph.nodes():
  amperage = 0
  # Wychodzace krawedzie
  for v2 in solution_graph[v1]:
   amperage -= solution_graph[v1][v2]["weight"]
  # Wchodzace krawedzie
  for v2, w in solution_graph.in_edges(v1):
   amperage += solution_graph[v2][v1]["weight"]
  if abs(amperage) > eps:
   return False

 return True
```

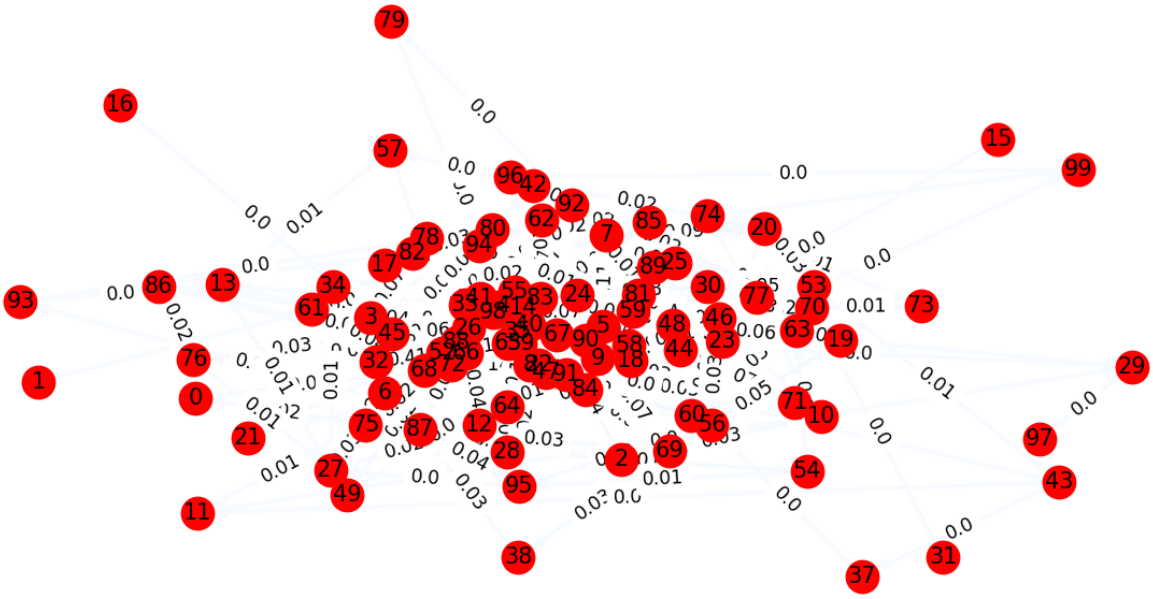
Powyższa funkcja sprawdza, czy rozwiązanie jest poprawne. Obliczane są zarówno spadki napięć w obwodach zamkniętych (drugie prawo Kirchhoffa), jak również sumy prądów wchodzących i wychodzących z węzłów.

- Graf losowy

```
def amperage_flow_random(n):
  directed_edges = create_random_graph(n)
  s, t, _ = directed_edges[randint(0, len(directed_edges) - 1)]
  stE = (s, t, randint(10, 20))
  solve_and_test(directed_edges, stE, draw_graph)


def create_random_graph(n):
  edges = []
  for i, j in nx.gnm_random_graph(n, 2 * n).edges():
    edges.append((i, j, randint(1, 10)))
  return edges
```

Poniżej przedstawiony jest przykładowe wywołanie dla 100 wierzchołków (parametry: ilość wierzchołków).

```
amperage_flow_random(100)
```



Rysunek 3. Przedstawia przepływ prądu w grafie spójnym (słabo widać krawędzie).

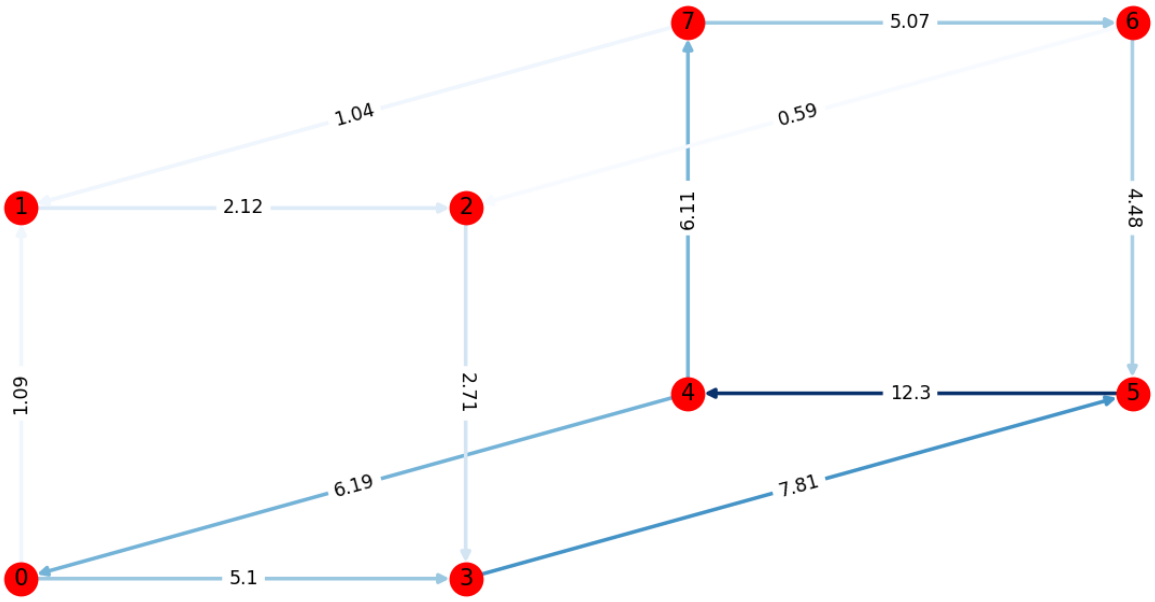
Widać wartości większe od 0, więc możemy przypuszczać, że jest wszystko ja należy.

- Graf 3-regularny (kubiczny)

```
def amperage_flow_cubical():
  directed_edges = create_cubical_graph()
  s, t, _ = directed_edges[randint(0, len(directed_edges) - 1)]
  stE = (s, t, randint(100, 500))
  solve_and_test(directed_edges, stE, draw_cubical_graph)


def create_cubical_graph():
  edges = []
  for i, j in nx.generators.small.cubical_graph().edges():
    edges.append((i, j, randint(2, 20)))
  return edges


def draw_cubical_graph(graph):
  plt.figure(figsize=(12, 6))
  labels = nx.get_edge_attributes(graph, 'weight')
  edges, weights = zip(*labels.items())
  pos = {0: (0, 0), 1: (0, 2), 2: (2, 2), 3: (2, 0), 4: (3, 1), 5: (5, 1), 6: (5, 3), 7: (3, 3)}
  nx.draw(graph, pos, with_labels=True, node_color='r', edgelist=edges, edge_color=weights,
width=2.0,
        edge_cmap=plt.cm.Blues)
  nx.draw_networkx_edge_labels(graph, pos=pos, edge_labels=labels)
  plt.show()
```

Teraz przetestujemy działanie.

```
amperage_flow_cubical()
```



Rysunek 4. Przedstawia przepływ prądu w grafie kubicznym.

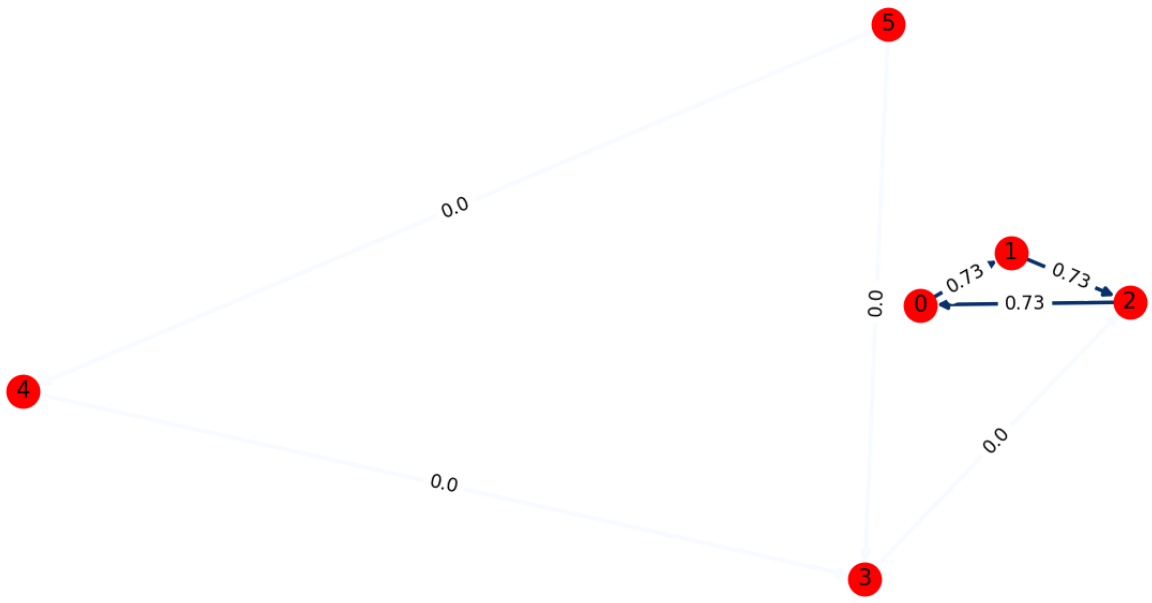
Wygląda w porządku.

- Graf złożony z dwóch grafów losowych połączonych mostkiem

```
def amperage_flow_bridge(n):
  s = randint(0, n - 2)
  t = randint(s + 1, n - 1)
  stE = (s, t, randint(10, 20))
  directed_edges = create_bridge_graph(n)
  solve_and_test(directed_edges, stE, draw_graph)


def create_bridge_graph(n):
  edges = []
  for i, j in nx.gnm_random_graph(n, 2 * n).edges():
    edges.append((i, j, randint(1, 10)))
  for i, j in nx.gnm_random_graph(n, 2 * n).edges():
    edges.append((i + n, j + n, randint(1, 10)))
  edges.append((n - 1, n, randint(1, 10)))
  return edges
```

Zobaczmy, czy udało się nam utrzymać satysfakcjonujące rozwiązanie (parametry: ilość wierzchołków po jednej stronie mostu).

```
amperage_flow_bridge(3)
```



Rysunek 5. Przedstawia przepływ prądu w dwóch grafach spójnych połączonych mostem.

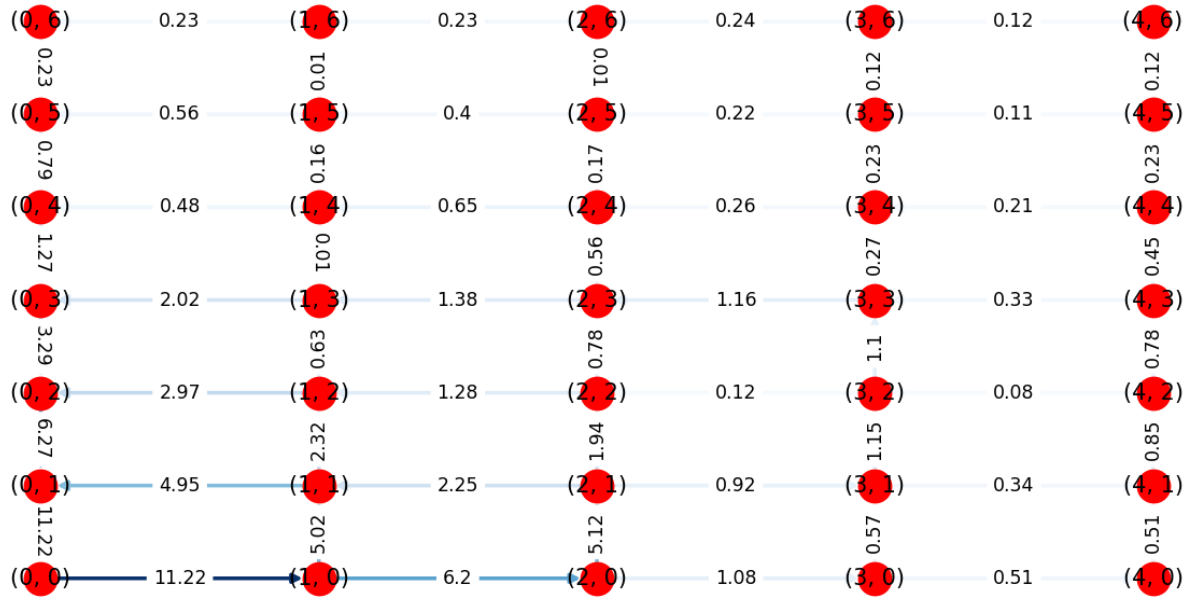
Wynik jest taki jak oczekiwaliśmy.

- Graf siatka 2D

```
def amperage_flow_net(n, k):
  directed_edges = create_net_graph(n, k)
  stE = (directed_edges[0][0], directed_edges[0][1], randint(100, 500))
  solve_and_test(directed_edges, stE, draw_net_graph)


def create_net_graph(n, k):
  edges = []
  for i, j in nx.generators.lattice.grid_2d_graph(n, k).edges():
    edges.append((i, j, randint(1, 10)))
  return edges


def draw_net_graph(graph):
  n = int(math.sqrt(graph.number_of_edges()))
  plt.figure(figsize=(12, 6))
  labels = nx.get_edge_attributes(graph, 'weight')
  edges, weights = zip(*labels.items())
  pos = {(i, j): (i, j) for i in range(n) for j in range(n)}
  nx.draw(graph, pos, with_labels=True, node_color='r', edgelist=edges, edge_color=weights,
width=2.0,
        edge_cmap=plt.cm.Blues)
  nx.draw_networkx_edge_labels(graph, pos=pos, edge_labels=labels)
  plt.show()
```

Teraz czas na przetestowanie powyższego kodu (parametry: rozmiar x siatki, rozmiar y siatki).

```
amperage_flow_net(5, 7)
```



Rysunek 6. Przedstawia przepływ prądu w grafie w postaci siatki 2D.

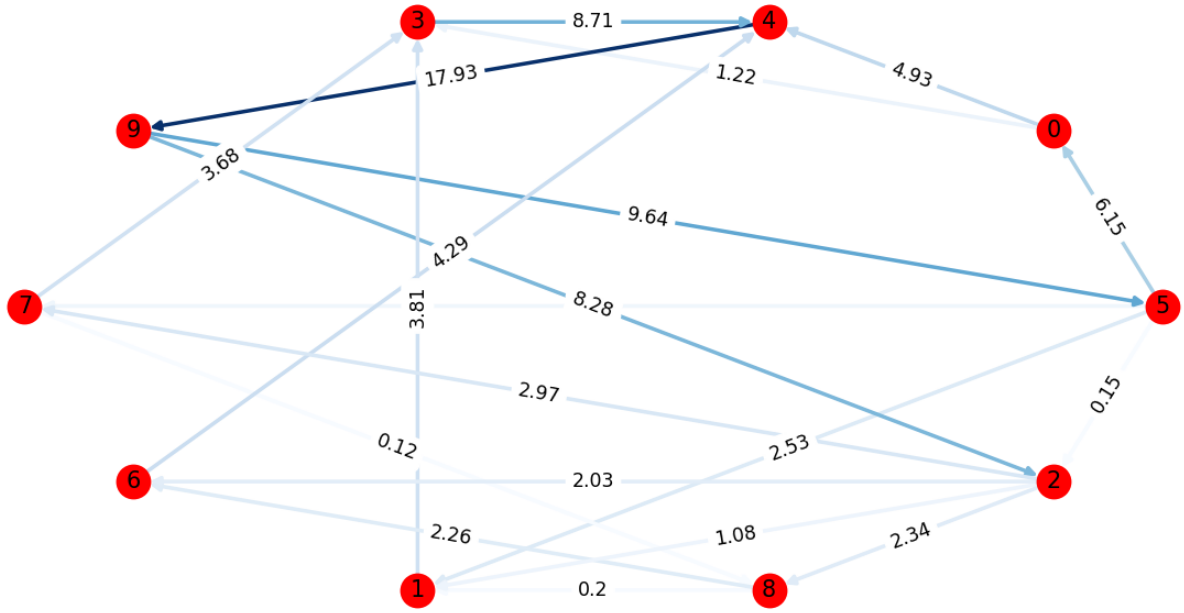
Prąd rozchodzi się tak jak powinien.

- Graf typu small-world

```
def amperage_flow_small_world(n, k, p):
  directed_edges = create_small_world_graph(n, k, p)
  s, t, _ = directed_edges[randint(0, len(directed_edges) - 1)]
  stE = (s, t, randint(100, 500))
  solve_and_test(directed_edges, stE, draw_small_world_graph)


def create_small_world_graph(n, k, p):
  edges = []
  for i, j in nx.watts_strogatz_graph(n=n, k=k, p=p).edges():
    edges.append((i, j, randint(1, 10)))
  return edges


def draw_small_world_graph(graph):
  plt.figure(figsize=(12, 6))
  labels = nx.get_edge_attributes(graph, 'weight')
  edges, weights = zip(*labels.items())
  pos = nx.circular_layout(graph)
  nx.draw(graph, pos, with_labels=True, node_color='r', edgelist=edges, edge_color=weights,
width=2.0,
        edge_cmap=plt.cm.Blues)
  nx.draw_networkx_edge_labels(graph, pos=pos, edge_labels=labels)
  plt.show()
```

Zobaczmy rezultat (parametry: ilość wierzchołków, ilość krawędzi do najbliższy sąsiadów, prawdopodobieństwo przepięcia).

```
amperage_flow_small_world(10, 4, 0.5)
```

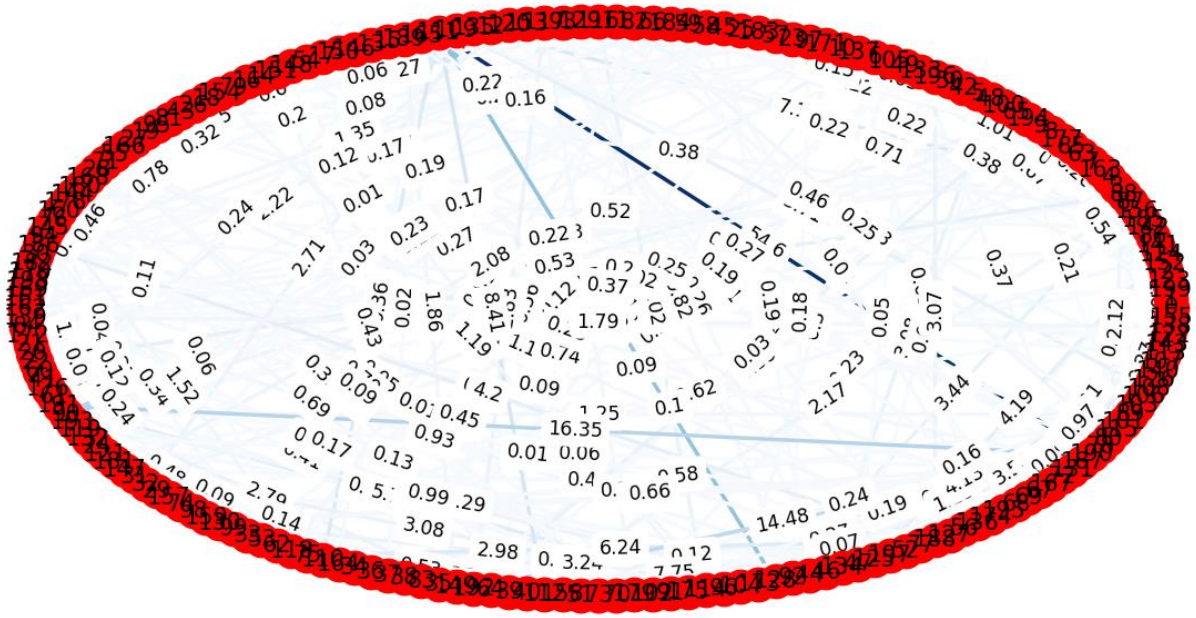


Rysunek 7. Przedstawia przepływ prądu w grafie typu small-world dla 10 wierzchołków.

Wygląda dobrze.

Teraz zobaczmy dla 200 wierzchołków.

```
amperage_flow_small_world(200, 4, 0.5)
```



Rysunek 8. Przedstawia przepływ prądu w grafie typu small-world dla 200 wierzchołków.

Bez problemu układ udało się rozwiązać i narysować przepływ prądu.

Należy zaznaczyć, że funkcja sprawdzająca poprawność rozwiązań za każdym razem wskazywała:

```
CORRECT: True
```

Rysunek 9. Przedstawia sprawdzenie poprawności dla powyższych układów.

co oznacza, że układy równań zostały rozwiązane poprawnie i wartości prądów są takie jak powinny.

e) Rozwiązanie przy pomocy praw Kirchhoffa:

Pierwsze prawo Kirchhoffa – w węźle suma algebraiczna natężeń prądów wpływających i wypływających jest równa 0.
W programie to prawo daje nam „n” układów równań, gdzie „n” jest liczbą węzłów (wierzchołków). Przy danej krawędzi ustawiamy 1 lub -1 jako współczynnik w zależności od tego, czy dana krawędź jest ustawiona jako wychodząca czy wchodząca. Wyraz wolny jest równy 0.

Drugie prawo Kirchhoffa – w zamkniętym obwodzie suma spadków napięć jest równa sumie sił elektromotorycznych występujących w tym obwodzie.
W programie oznacza to, że szukamy cyklów prostych, a następnie jeden cykl odpowiada jednemu równaniu. W wierszu (równaniu) współczynnikami będą wartości oporów elektrycznych ze znakiem plus bądź minus w zależności, w którą stronę jest skierowana krawędź. Jeśli cykl zawiera krawędź, która dostarcza siły elektromotorycznej to wyraz wolny jest ustawiany na jej wartość z odpowiednim znakiem.

Stosując oba prawa możemy otrzymać układ nadokreślony. Żeby sprawdzić, czy ma on rozwiązanie można zastosować metodę Gaussa-Jordana, a następnie sprawdzić czy dodatkowe równania (wiersze, które sprawiają, że macierz współczynników nie jest kwadratowa) są wyzerowane (każdy współczynnik i wyraz wolny). Jeśli tak jest to układ można rozwiązać, w przeciwnym wypadku nie ma on rozwiązania.

Kierunek prądu wyznaczany jest na podstawie rozwiązań układu równań. Jeśli wartość jest ujemna to oznacza, że trzeba odwrócić krawędź względem początkowego kierunku, inaczej kierunek jest ten sam.
Cykle proste wyznaczane są za pomocą funkcji z biblioteki „networkx".
Rozwiązanie weryfikowane jest przy pomocy wzoru $U = R * I$, który stosujemy dla drugiego prawa Kirchhoffa. Pierwsze prawo Kirchhoffa natomiast sprawdzamy przez sumę natężeń wchodzących i wychodzących z danego węzła. W obu przypadkach oczekujemy, że ta wartości te będą bliskie 0, wtedy rozwiązanie można uznać za poprawne.