Paweł Prus

IDE: Microsoft Visual Studio Code
Procesor: AMD Ryzen 7 6800h
Język: Python 3.10.12
Biblioteki:
import numpy jako np
import networkx jako nx
import matplotlib.pyplot jako plt
import matplotlib.patches jako mpatches
import matplotlib.colors jako mcolors
from collections import deque, OrderedDict

Znajdowanie przecięć odcinków

1. Generowanie grafów

1.1. W kodzie zawarta jest funkcja z pliku *zad3_gen.py create_graphs()* generująca 5 rodzajów grafów w postaci krawędziowej z pomocą następujących funkcji pomocniczych

1.1.1. *generate_connected_erdos_renyi_graph(n,p)*, generująca spójny graf Erdosa_Renyi'ego o *n* wierzchołkach z prawdopodobieństwem *p* na to, że między wierzchołkami pojawi się krawędź

1.1.2. *generate_connected_3_regular_graph(n)*, która generuje spójny graf 3-regularny

1.1.3. *generate_2_random_connected(n, p, i, text, directory),* generująca dwa losowe grafy Erdosa_Renyi'ego i łącząca je jedną losowo wygenerowaną krawędzią

1.1.4. *generate_2D_grid_graph(n),* która generuje graf typu siatka 2D o n wierzchołkach

1.1.5. *generate_small_world_graph(n, k, p)*, która generuje spójny graf typu small-world

1.2. Wszystkie grafy zapisywane są w postaci krawędziowej do plików *.txt*, a ich obrazy generowane z pomocą biblioteki *networkx* języka Python są eksportowane do plików *.pdf*

2. Rozwiązanie

2.1.1. Rozwiązanie zawiera się w pliku *zad3.py* w funkcji *check_graphs(text,numer)*.

2.1.2. Argumenty funkcji, text oraz numer, pozwalają na dokładną identyfikację jednego z wygenerowanych grafów z punktu **1**.

2.1.3. Następnie funkcja znajduje nazwę pliku z grafem *filename* odczytuje graf zapisany w pliku i tworzy przy użyciu funkcji *read_graph(filename)* obiekt typu *nx.DiGraph()*. Potem inicjalizowane są tablice *to_solve* oraz *vector*, które są następnie wypełniane pierwszymi danymi.

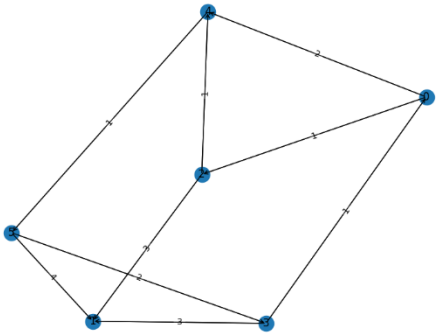
2.1.4. Rozwiązanie opiera się na metodzie **I** oraz **II** prawa Kirchoffa, szukając odpowiedniej liczby ścieżek z wierzchołka *s* do *t*, będących wierzchołkami kolejno startowymi oraz końcowymi, tzn. do *s* i *t* przykładana jest siła elektromotoryczna E. Pierwszym krokiem jest ułożenie równań dla wszystkich węzłów (wierzchołków), które pokazują jakie natężenia "wchodzą" do wierzchołka a jakie z niego "wychodzą", oraz jakie są między nimi relacje

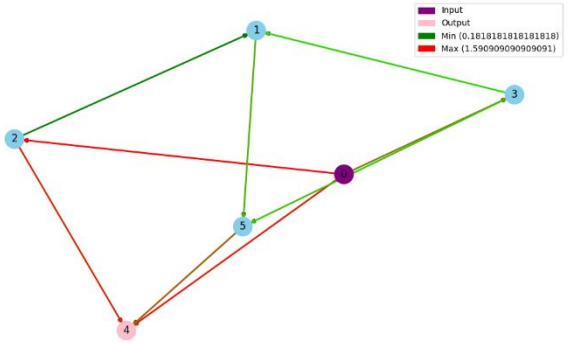
2.1.5. Następnie funkcja *check_graphs* wywołuje funkcję *find_paths*, która na bieżąco przeszukuje graf szukając kolejnych możliwych do zrealizowania ścieżek pomiędzy wierzchołkami *s* oraz *t*, które razem z podłączonym ogniwem utworzą ”oczka” układu.

2.1.6. Gdy tylko uzbiera się wystarczająca liczba równań do rozwiązania układul, podejmowana jest ku temu próba. Jeśli *np.linalg.solve()* znajdzie rozwiązanie, jest ono zwracane. W przeciwnym przypadku, tj. gdy podniesie *Exception(”Singular Matrix”)*, najwcześniejsze równanie powstałe ze ścieżki i znajdujące się w macierzy *to_solve* zostaje usunięte z macierzy, gdzie zostanie zastąpione przez nowe równanie powstałe z nowo znalezionej ścieżki.

2.1.7. Gdy zostanie zwrócone rozwiązanie, wartości natężeń są przekazywane do nowego grafu skierowanego, który z kolei zostaje narysowany i wyeksportowany do pliku, a kolory jego krawędzi oznaczają jakie natężenie prądu tamtędy płynie. Im bardziej zielona jest krawędź, tym mniejsze natężenie, a im bardziej czerwona tym większe.

2.1.8. Poniżej przykładowe rozwiązanie zadania dla grafu 3-regularnego





3. Wnioski

3.1. Algorytm wydaje się być powolny, jednak jest to zapewne spowodowane błędami implementacyjnymi, które da się naprawić. Program zdaje nie radzić sobie z niektórymi przykładami, które z jakiegoś powodu nie wpisują się w ogólny schemat wyglądu przykładu, więc ten problem także należy rozpatrzeć.