Metody Obliczeniowe w Nauce i Technice
Laboratorium II

Maciej Trątnowiecki

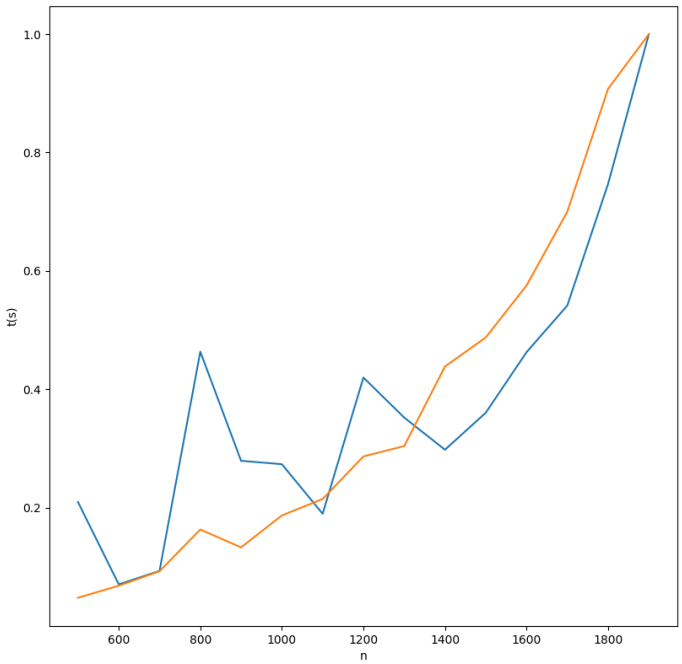
AGH, Semestr Letni, 2020

Metoda Gaussa-Jordana

W ramach rozwiązania zadania zaimplementowałem funkcję rozwiązującą układ równań liniowych metodą Gaussa-Jordana. Zastosowałem normalizację (skalowanie) wartości wierszy i częściowe poszukiwanie elementu wiodącego. Otrzymałem rozwiązanie zwracające poprawne wyniki dla testowanych losowych macierzach. Jednakże moja implementacja w pythonie nie jest równie wydajna co choćby biblioteczna funkcja z pakietu numpy. Dla macierzy o rozmiarach nxn wykonałem pomiary czasu wykonania, otrzymując następujące wyniki.

n	Czas numpy	Mój czas	Mój czas jako procent numpy
500	0.0716	2.5172	3516%
1000	0.0689	7.6921	11164%
2000	0.1845	38.7065	20979%

Różnica jest zatrważająca, nie jest to jednak dużym zaskoczeniem. Funkcje z pakietu numpy z natury są wydajniejsze od zwykłych funkcji pythonowych. Postanowiłem jednak zbadać wzrost czasu potrzebnego na wykonanie obliczeń w zależności od liczby elementów. Otrzymane wartości znormalizowałem, by móc porównać tempo wzrostu.



(a)

Rysunek 1: Wykres zależności czasu wykonania od rozmiaru układu.

Otrzymana zależność sugeruje, że oba algorytmy działają w porównywalnej złożoności obliczeniowej, różnią się za to zdecydowanie w czasach wykonania.

Faktoryzacja LU

Następnie przygotowałem implementację prostej faktoryzacji LU macierzy, korzystając z algorytmu Gaussa. Zastosowałem przygotowaną wcześniej normalizację macierzy i częściowe poszukiwanie elementu wiodącego. Poprawność faktoryzacji sprawdziłem mnożąc ze sobą wynikowe macierze L i U.

```
Before decomposition:
[[ 0.98728914 -0.39653349 -0.79527545  0.75688783 -0.94968234 -0.3346223
  -0.81891081 -0.0872668   0.03600341  1.        ]
 [-0.24875487  0.7988324   0.89159502 -0.41250366  1.         -0.93530562
   0.07012895 -0.41455246  0.19522191  0.06766288]
 [-0.43676251  1.         -1.07636731 -1.28462965  0.45643018 -1.17210551
   0.81369935 -1.05006032  0.89648988 -0.44792026]
 [-0.18849563  1.         -0.33129675 -0.77867599 -0.20658434  0.56841036
  -0.48853016 -0.35456374 -0.13448072 -0.69259971]
 [-0.54178984  0.69162536  0.76137258 -0.1303646   1.         -0.73369476
  -0.31902962 -0.01001028 -0.24317943  0.95656748]
 [-0.58160768 -0.01424781  1.          0.74573179  0.78382316  0.33646435
  -0.06520431 -0.86223762 -0.21060048  0.41415808]
 [ 0.0765054   0.08652725  0.13080835  1.          0.70916827  0.41925449
   0.08555159 -0.24048492 -0.93408119  0.24964034]
 [ 0.39831885 -1.04714032  0.02959505 -0.03311721 -1.07846632 -1.0795346
  -0.01333096  1.         -0.32202346 -1.25990451]
 [ 0.95970704  0.78651847  0.50651952  1.          0.95091278  0.73772133
  -0.82095734 -0.71323635 -0.45481701  0.12017433]
 [ 0.51354035  0.69346936  0.84536493  0.36391358  1.         -0.56789755
  -0.28757392 -0.48314493 -0.74834116 -0.66201069]]
```

(a)

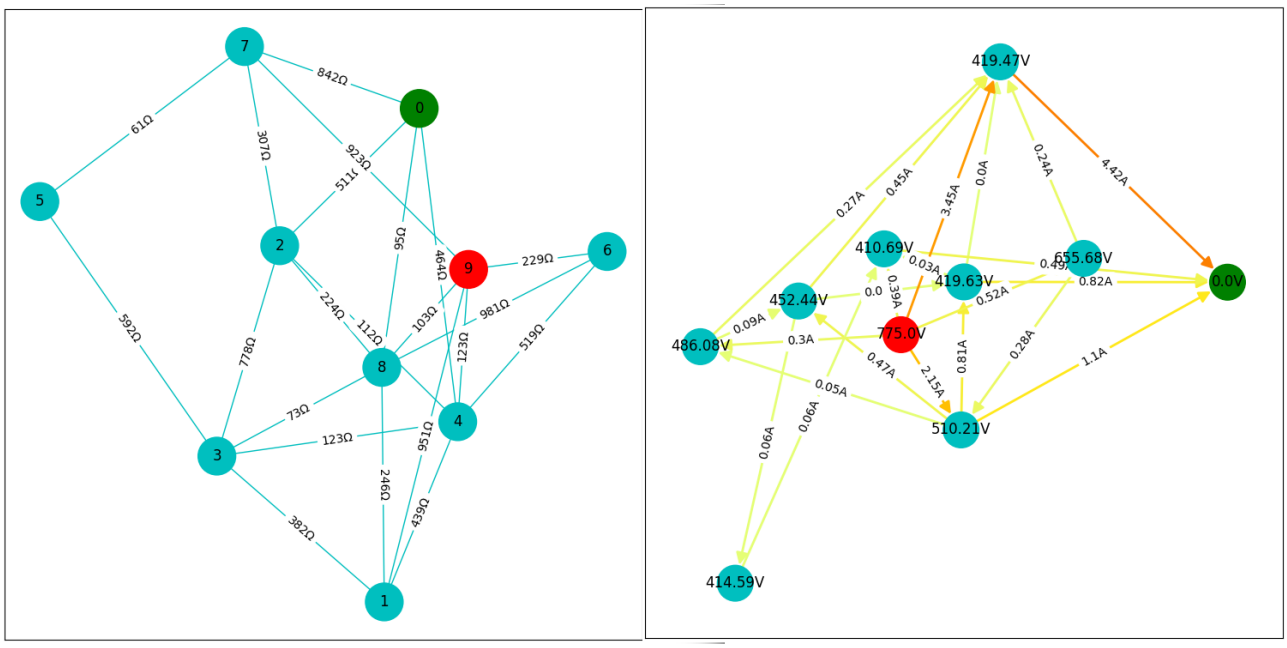
```
Matrix L:
[[  1.          0.          0.          0.          0.
    0.          0.          0.          0.          0.        ]
 [ -0.25195747  1.          0.          0.          0.
    0.          0.          0.          0.          0.        ]
 [ -0.44238561  1.1797859   1.          0.          0.
    0.          0.          0.          0.          0.        ]
 [ -0.19092242  1.3224534   0.62274488  1.          0.
    0.          0.          0.          0.          0.        ]
 [ -0.54876512  0.6782174   0.06411079  5.46960828  1.
    0.          0.          0.          0.          0.        ]
 [ -0.58909559 -0.35460845 -0.34613677  9.97779025  1.85883376
    1.          0.          0.          0.          0.        ]
 [  0.07749037  0.16776499 -0.0340834  10.89403161  2.11696489
    4.71579682  1.          0.          0.          0.        ]
 [  0.40344701 -1.2693248  -0.54723951 -11.36712224 -2.112917
   -6.16591064 -1.54137769  1.          0.          0.        ]
 [  0.97206279  1.6768288  -0.05371565  6.83471656  1.36203798
    8.37040618  2.28387467 -1.63126556  1.          0.        ]
 [  0.52015193  1.28730525 -0.16455962  1.62542853  0.37523782
    3.55367746  0.95852278 -0.57103274  0.93116249  1.        ]]
Matrix U:
[[  0.98728914 -0.39653349 -0.79527545  0.75688783 -0.94968234
   -0.3346223  -0.81891081 -0.0872668   0.03600341  1.        ]
 [  0.         0.69892282  0.69121943 -0.22180012  0.76072044
   -1.0196162  -0.13620174 -0.43653999  0.20429324  0.31962034]
 [  0.         0.         -2.24367666 -0.68811671 -0.86118288
   -0.11720878  0.61211389 -0.57364218  0.67139498 -0.38261823]
 [  0.         0.          0.          0.08767236 -0.8576201
    1.92590954 -0.84594892  0.56331159 -0.81588294 -0.68608676]
 [  0.         0.          0.          0.          4.70897075
  -10.75225879  3.91172123 -2.8061473   4.05753916  5.06571633]
 [  0.         0.          0.          0.          0.
    0.50754527  0.78541722 -1.67144901  0.71386559 -1.58653921]
 [  0.         0.          0.          0.          0.
    0.         -2.57632808  7.5059569  -4.01611944  4.33759183]
 [  0.         0.          0.          0.          0.
    0.          0.          1.90476143 -2.19954368 -1.65901901]
 [  0.         0.          0.          0.          0.
    0.          0.          0.         -1.13758017 -2.95171815]
 [  0.         0.          0.          0.          0.
    0.          0.          0.          0.          0.83930467]]
LU decomposition correct:  True
```

(b)

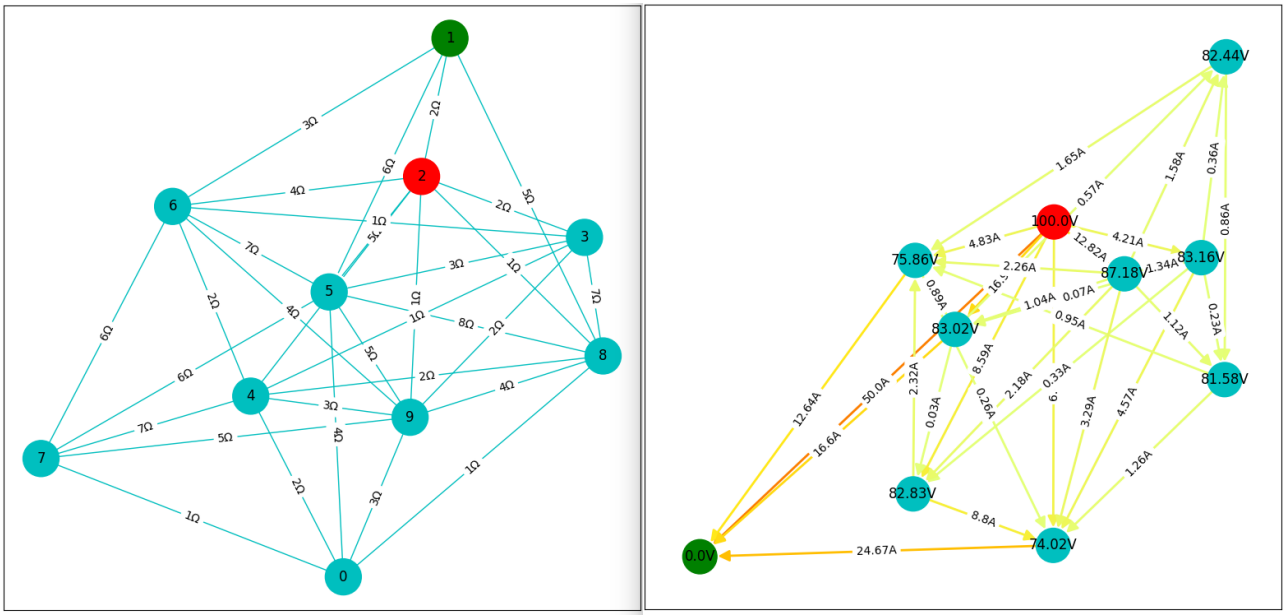
Rysunek 2: Przykładowy wynik działania programu.

Analiza obwodu elektrycznego

Przygotowałem program analizujący natężenie prądu przepływającego w zadanym obwodzie elektrycznym. Obwód przyjmuje jako graf ważony, nieskierowany o wagach krawędzi oznaczających ich rezystancję. Dodatkowo w grafie wyróżnione zostały dwa węzły, pomiędzy którymi przyłożono znaną siłę elektromotoryczną.
Wynikiem analizy natężeń jest ważony graf skierowany, gdzie wagami krawędzi są natężenia płynącego nimi prądu. Dodatkowo w węzłach grafu oznaczam jego napięcie względem wierzchołka źródłowego.
Do implementacji programu wykorzystałem bibliotekę *networkx*, reprezentując powyżej określone struktury jako pochodne klas *nx.Graph* i *nx.DiGraph*. Dodatkowo przygotowałem wizualizację obu grafów przy wykorzystaniu biblioteki *matplotlib*. W analizie natężeń wykorzystałem także funkcję *numpy.linalg.solve* odpowiadającą za rozwiązanie układu równań liniowych.
Zaimplementowałem także funkcję sprawdzającą poprawność uzyskanych wyników. Za poprawny uznaje się graf wynikowy, w którym suma natężeń prądów wychodzących z węzła, jest równa sumie natężeń prądów wchodzących.
Program umożliwia wczytanie układu z pliku, lub wygenerowanie losowego przykładu. Przygotowałem zbiór przykładowych układów dla celów testowych, znajdujących się w folderze graphs. Funkcja losująca umożliwia wybór typu generowanego grafu, w tym graf regularny i graf z pojedynczym mostkiem. Ze względu na wydajność funkcji generującej wizualizację, dla grafów o dużej liczbie wierzchołków (rzędu kilku tysięcy) warto ograniczyć się do sprawdzenia poprawności wyniku.

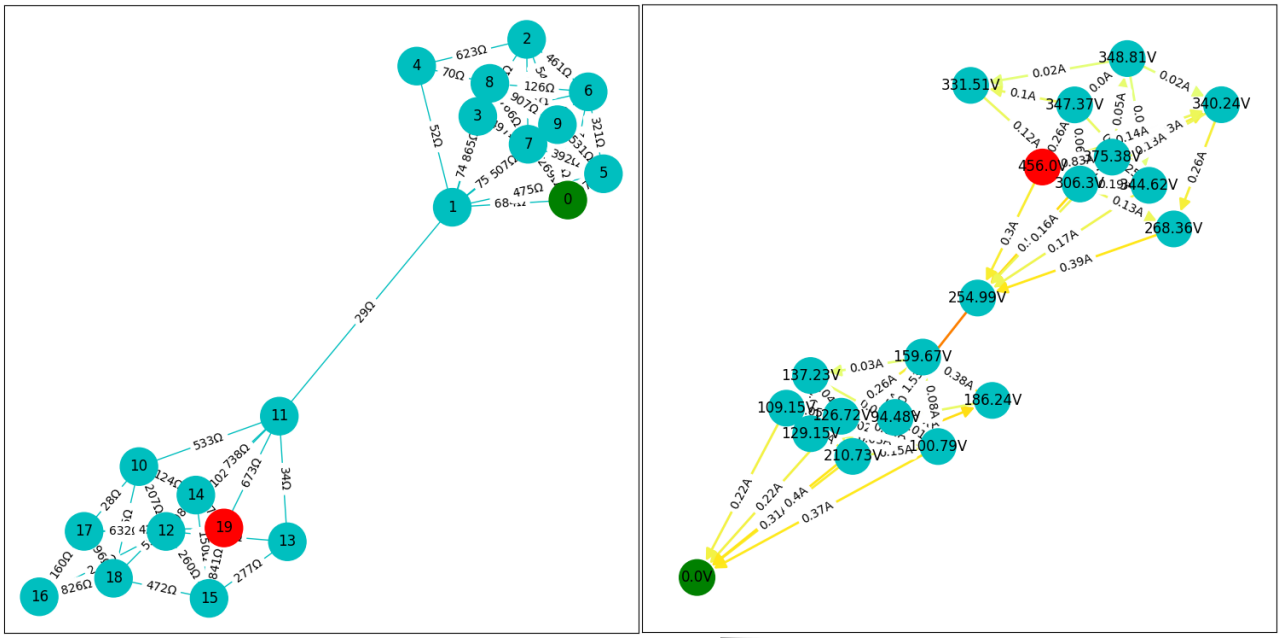


(a) Losowy graf Erdosza o 10 wierzchołkach

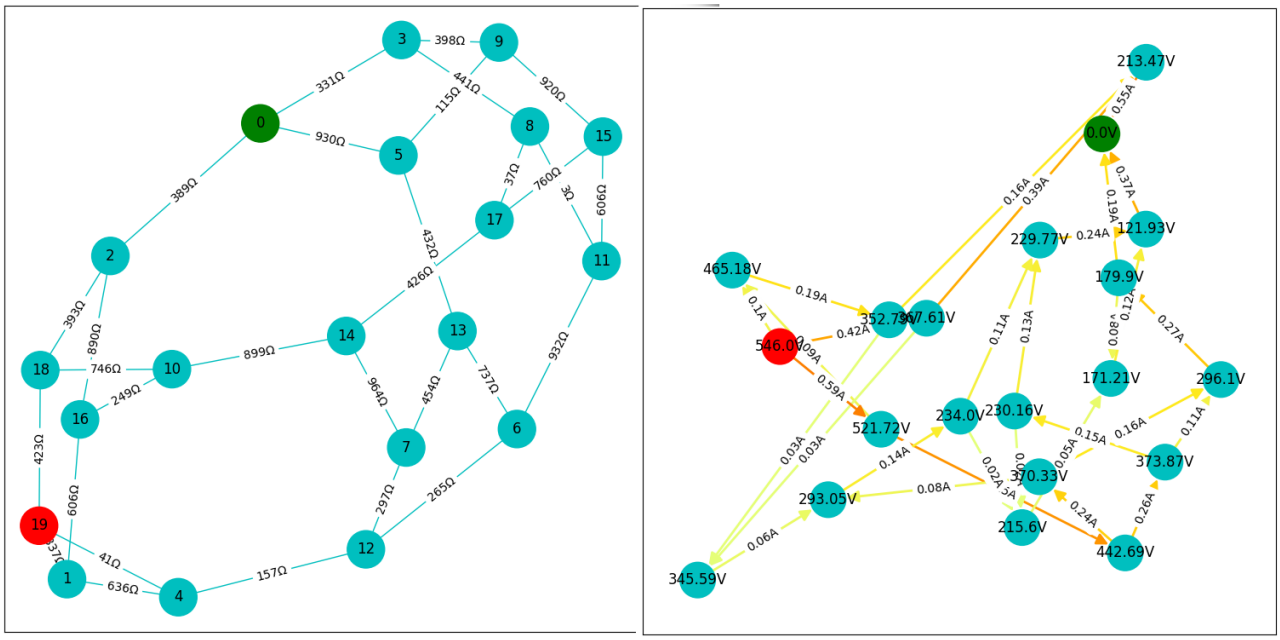


(b) Graf o 10 wierzchołkach

Rysunek 3: Przykładowy wynik działania programu.



(a) Graf o jednym mostku i 20 wierzchołkach



(b) Graf 3-regularny o 20 wierzchołkach

Rysunek 4: Przykładowy wynik działania programu.

Program zwraca poprawne wyniki dla dużo większych grafów. Czas wykonania obliczeń (bez wizualizacji) dla losowego grafu o 1000 wierzchołków wyniósł 8,6 sekund. Dla grafu Erdosza o tej samej liczbie węzłów 9,2 sekund.