TEORIA WSPÓŁBIEŻNOŚCI

SPRAWOZDANIE: WSPÓŁBIEŻNE MNOŻENIE METODĄ GAUSSA Z WYKORZYSTANIEM TEORII ŚLADÓW

Aleksandra Smela

SPIS TREŚCI

1. OPIS ZADANIA

1.1. CZĘŚĆ TEORETYCZNA

Dla macierzy o rozmiarze *NxN* należy:

- Zlokalizować niepodzielne czynności wykonywane przez algorytm eliminacji Gaussa, nazwać je i zbudować alfabet w sensie teorii śladów.
- Przedstawić algorytm sekwencyjny w postaci ciągu symboli alfabetu.
- Skonstruować relację zależności i niezależności dla alfabetu, opisującego algorytm.
- Wygenerować graf zależności Diekerta.
- Przekształcić ciąg symboli opisujący algorytm do postaci normalnej Foaty.

1.2. CZĘŚĆ IMPLEMENTACYJNA

Dla macierzy o rozmiarze *NxN* zadanej na wejściu:

- Zaprojektować i zaimplementować równoległy algorytm eliminacji Gaussa bazujący na informacji uzyskanej z grafu zależności. Rozwiązanie może nie uwzględniać możliwości zamiany miejscami wierszy.

2. UŻYTE NARZĘDZIA I BUDOWA PROJEKTU

Projekt składa się z trzech plików: ***graph_generation.py***, *get_classes.py* oraz ***GaussianElimination.java***, z czego ***graph_generation.py*** i ***GaussianElimination.java*** to pliki wykonywalne. ***graph_generation.py*** realizuje część teoretyczną i generuje graf. ***GaussianElimination.java*** realizuje część implementacyjną i współbieżną eliminację Gaussa. Szczegóły dotyczące użytych narzędzi i uruchomienia zawarte są w punktach 2.1 oraz 2.2.

2.1. CZĘŚĆ TEORETYCZNA

Część teoretyczna została napisana w języku Python z użyciem bibliotek *graphviz*, *matplotlib* i *queue*. Program znajduje się w pliku *graph_generation.py.* Jest uruchamiany z jednym argumentem będącym ścieżką do pliku .txt, który zawiera dane wejściowe – macierz *NxN* z dodatkową kolumną na wyrazy wolne (notacja taka jak w punkcie 3.1).

Przykład poprawnego pliku .txt:

```
1, 2, 3 / 3
5, 6, 7 / 5
9, 10, 6 / 12
```

Przykład poprawnego uruchomienia programu *graph_generation*:

```
python graph_generation.py examples/example.txt
```

Program wypisuje na standardowe wyjście alfabet, algorytm sekwencyjny, zależności i postać normalną Foaty i tworzy pliki w formacie .pdf i .gv zawierające graf Diekerta.

2.2. CZĘŚĆ IMPLEMENTACYJNA

Współbieżna eliminacja Gaussa została napisana w języku Java z użyciem *CountDownLatch* i *ConcurrentHashMap* z pakietu *java.util.concurrent* oraz *ProcessBuilder*, który posłużył do uruchomienia skryptu *get_classes.py* napisanego w Pythonie. Skrypt wykorzystuje metodę z *graph_generation.py*, aby uzyskać postać normalną Foaty, którą wykorzystuję potem do współbieżnego uruchomienia wątków. Implementacja znajduje się w pliku *GaussianElimination.java*.

Program jest uruchomiany z jednym argumentem będącym ścieżką do pliku .txt, który zawiera dane wejściowe w takim samym formacie jak w punkcie 2.1. Program wypisuje na standardowe wyjście macierz oryginalną i macierz po uruchomieniu współbieżnej eliminacji Gaussa.

Przykład poprawnego uruchomienia programu *GaussianElimination*:

```
java GaussianElimination.java examples/example.txt
```

3. WPROWADZENIE TEORETYCZNE

3.1. ALFABET

Dla układu równań postaci:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Zastosujmy notację:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & | & a_{1(n+1)} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} & | & a_{2(n+1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & | & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} & | & a_{n(n+1)} \end{bmatrix}$$

gdzie $a_{i(n+1)} = b_i$.

Algorytm eliminacji Gaussa realizuje następujące operacje:

- Znalezienie mnożnika dla wiersza *i*, który będzie odejmowany od wiersza *k*.

$$A_{i,k}: \quad m_{k,i} = \frac{M_{k,i}}{M_{i,i}}$$

- Pomnożenie *j*-tego elementu wiersza *i*-tego przez mnożnik żeby następnie odjąć tę wartość od *k*-tego wiersza.

$$B_{i,j,k}: \quad n_{k,i} = M_{i,j} \cdot m_{k,i}$$

- Odjęcie *j*-tego elementu wiersza *i* od wiersza *k*.

$$C_{i,j,k}: \quad M_{k,j} = M_{k,j} - n_{k,i}$$

Zdefiniujmy zatem alfabet Σ postaci:

$$\Sigma = \{A_{i,k}, B_{i,j,k}, C_{i,j,k}\}$$

gdzie: $i = \{1, 2, \dots, N\}$, $k = \{i+1, \dots, N\}$, $j = \{i+1, \dots, N+1\}$, $N -$ rozmiar macierzy

3.2. ALGORYTM SEKWENCYJNY

Algorytm sekwencyjny eliminacji Gaussa dla macierzy

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & | & a_{1(n+1)} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} & | & a_{2(n+1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & | & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} & | & a_{n(n+1)} \end{bmatrix}$$

składa się z następujących kroków:

- Wyzerowanie pierwszego elementu w drugim wierszu:

$$A_{1,\,2}, B_{1,1,2}, C_{1,1,2}, \dots, B_{1,\,i,2}, C_{1,i,\,2}, \dots, B_{1,\,(n+1),2}, C_{1,\,(n+1),2}$$

- Wyzerowanie pierwszego elementu w trzecim wierszu:

$$A_{1,\,3}, B_{1,1,3}, C_{1,1,3}, \dots, B_{1,\,i,3}, C_{1,i,\,3}, \dots, B_{1,\,(n+1),3}, C_{1,\,(n+1),3}$$

- Wyzerowanie drugiego elementu w trzecim wierszu:

$$A_{2,\,3}, B_{2,2,3}, C_{2,2,3}, \dots, B_{2,\,i,3}, C_{2,i,\,3}, \dots, B_{2,\,(n+1),3}, C_{2,\,(n+1),3}$$

Następnie wyzerowanie pierwszego, drugiego i trzeciego elementu w czwartym wierszu i tak dalej.

Można to zapisać bardziej ogólnie. Dla kolejnych wierszy $k = \{2, 3, \dots, N\}$ wykonujemy zerowanie *m*-tego elementu dla $m = \{1, 2, \dots, k-1\}$ następującymi operacjami:

$$A_{m,\,k}, B_{m,m,k}, C_{m,m,k}, \dots, B_{m,\,i,k}, C_{m,i,\,k}, \dots, B_{m,\,(n+1),k}, C_{m,\,(n+1),k}$$

3.3. RELACJA ZALEŻNOŚCI

Można zauważyć że zależności między operacjami są następujące:

- $A_{i,\,k}$ jest zależne od $C_{i-1,i,\,i}$ oraz $C_{i-1,\,i,k}$
- $B_{i,j,\,k}$ jest zależne od $A_{i,k}$ oraz $C_{i-1,\,j,k-1}$
- $C_{i,j,\,k}$ jest zależne od $B_{i,j,k}$ oraz $C_{i-1,\,j,k}$

Zatem:

$$D = sym\{(A_{i,\,k}, C_{i-1,i,\,i}), (A_{i,\,k}, C_{i-1,\,i,k}), (B_{i,j,\,k}, A_{i,k}), (B_{i,j,\,k}, A_{i,k}), (C_{i,j,\,k}, B_{i,j,k}), (C_{i,j,\,k}, C_{i-1,\,j,k})\} \cup I_\Sigma$$

gdzie: $i = \{1, 2, \dots, N\}$, $k = \{\, i+1, \dots, N\}$, $j = \{i+1, \dots, N+1\}$, $N -$ rozmiar macierzy

3.4. GRAF ZALEŻNOŚCI DIEKERTA I POSTAĆ NORMALNA FOATY

Aby stworzyć graf zależności Diekerta najpierw stworzyłam pełny graf zależności na podstawie relacji opisanej w punkcie 3.3. Następnie usunęłam nadmiarowe krawędzie, czyli takie krawędzie *(A, B)*, że z wierzchołka *A* do wierzchołka *B* istnieje ścieżka dłuższa niż 1.

Mając wspomniany graf, można łatwo sprowadzić sekwencję opisaną w 3.2 do postaci normalnej Foaty. Wystarczy uruchomić zmodyfikowany algorytm BFS na grafie. Na początku na kolejkę odkładam wszystkie wierzchołki grafu, do których nie prowadzi żadna krawędź. Co istotne, krawędzie są skierowane, więc z wierzchołków początkowych mogą (a nawet powinny) wychodzić krawędzie. Wierzchołki początkowe przypisuję do klasy 0. Ponadto modyfikacja BFS pozwala na ponowne odwiedzanie wierzchołków, z tym, że przy każdej kolejnej wizycie zwiększana jest klasa takiego ponownie odwiedzanego wierzchołka i wynosi *klasa rodzica + 1*.

Na podstawie obliczonych klas można pokolorować graf.

4. WYNIK DZIAŁANIA PROGRAMU DLA CZĘŚCI TEORETYCZNEJ

Wszystkie kroki opisane we Wprowadzeniu teoretycznym (3) zostały zaimplementowane. Program *graph_generation.py* wyznacza alfabet, kolejne kroki algorytmu sekwencyjnego, zależności, postać normalną Foaty oraz rysuje graf Diekerta wraz z kolorowaniem.

Szczegóły dotyczące uruchomienia programu zostały opisane w punkcie 2.1.

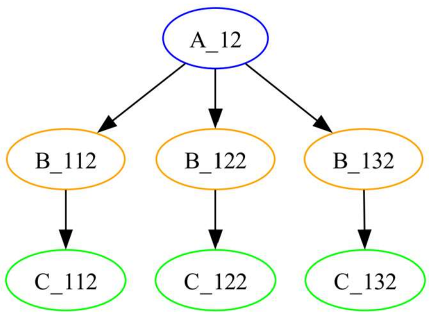
4.1. WYNIK DZIAŁANIA DLA MACIERZY 2X2

Alfabet: A_12, B_112, B_122, B_132, C_112, C_122, C_132

Algorytm sekwencyjny: A_12, B_112, C_112, B_122, C_122, B_132, C_132

Zależności: ('C_112', 'B_112'), ('B_122', 'A_12'), ('C_132', 'B_132'), ('B_112', 'A_12'), ('B_132', 'A_12'), ('C_122', 'B_122')

FNF: ('A_12',), ('B_112', 'B_122', 'B_132'), ('C_112', 'C_122', 'C_132')



Rys. 1: Graf dla macierzy 2x2

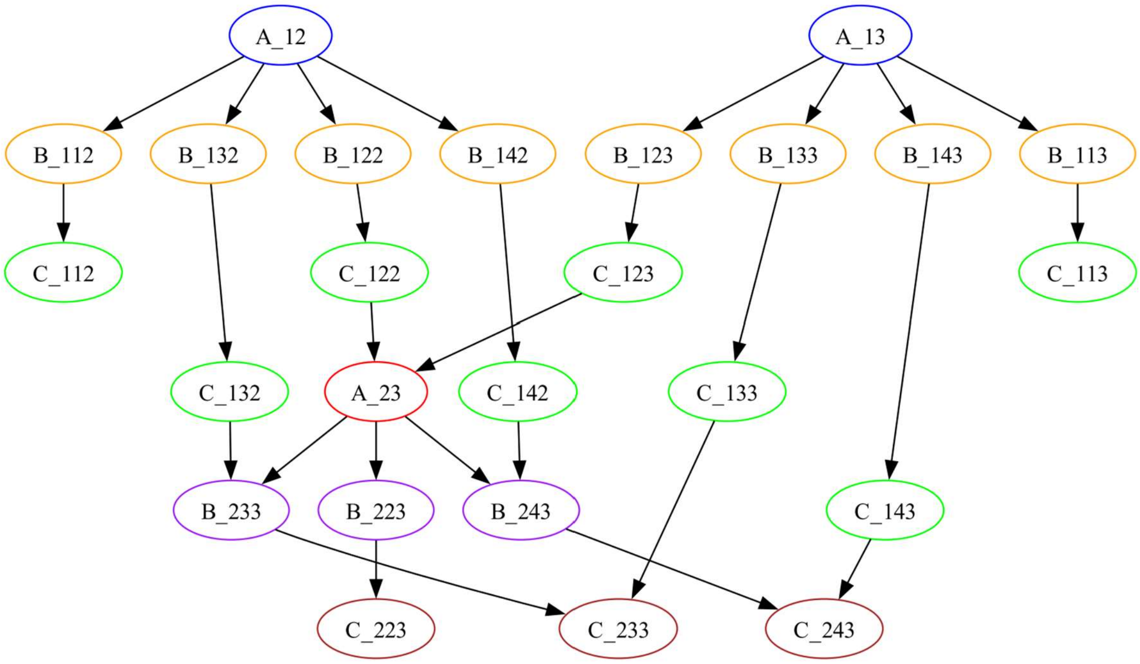
4.2. WYNIK DZIAŁANIA DLA MACIERZY 3X3

Alfabet: A_12, A_13, A_23, B_112, B_113, B_122, B_123, B_132, B_133, B_142, B_143, B_223, B_233, B_243, C_112, C_113, C_122, C_123, C_132, C_133, C_142, C_143, C_223, C_233, C_243

Algorytm sekwencyjny: A_12, B_112, C_112, B_122, C_122, B_132, C_132, B_142, C_142, A_13, B_113, C_113, B_123, C_123, B_133, C_133, B_143, C_143, A_23, B_223, C_223, B_233, C_233, B_243, C_243

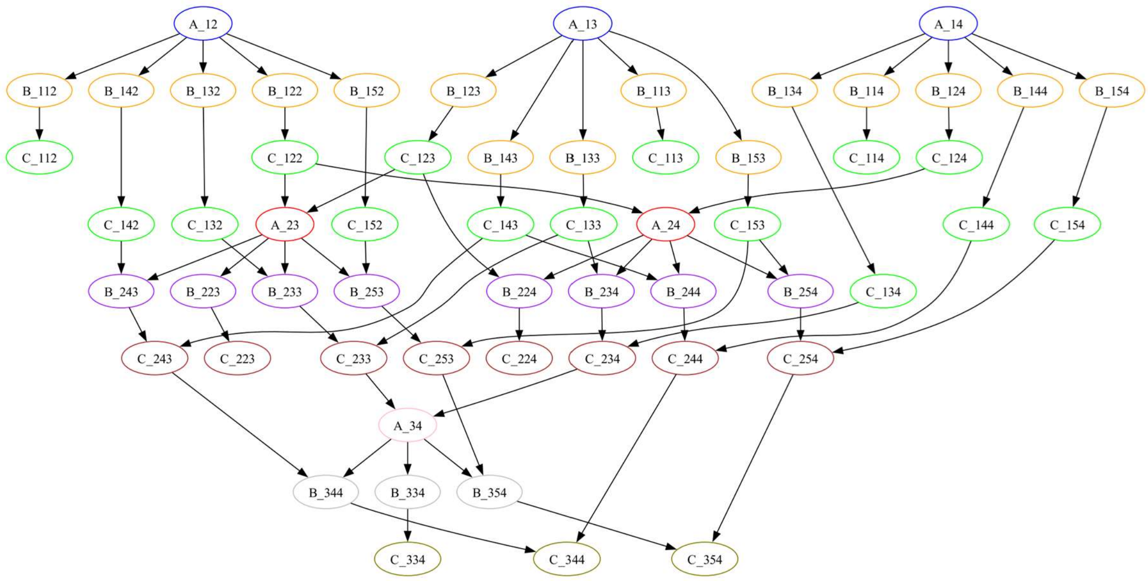
Zależności: : ('B_233', 'C_132'), ('B_112', 'A_12'), ('B_243', 'C_142'), ('B_132', 'A_12'), ('B_143', 'A_13'), ('B_243', 'A_23'), ('B_133', 'A_13'), ('C_243', 'B_243'), ('B_122', 'A_12'), ('A_23', 'C_122'), ('B_123', 'A_13'), ('C_233', 'B_233'), ('C_243', 'C_143'), ('C_123', 'B_123'), ('B_142', 'A_12'), ('C_133', 'B_133'), ('C_132', 'B_132'), ('B_233', 'A_23'), ('C_223', 'C_123'), ('C_223', 'B_223'), ('B_113', 'A_13'), ('C_112', 'B_112'), ('C_233', 'C_133'), ('A_23', 'C_123'), ('C_122', 'B_122'), ('C_143', 'B_143'), ('B_223', 'C_122'), ('B_223', 'A_23'), ('C_142', 'B_142'), ('C_113', 'B_113')

FNF: ('A_12', 'A_13'), ('B_112', 'B_113', 'B_122', 'B_123', 'B_132', 'B_133', 'B_142', 'B_143'), ('C_112', 'C_113', 'C_122', 'C_123', 'C_132', 'C_133', 'C_142', 'C_143'), ('A_23',), ('B_223', 'B_233', 'B_243'), ('C_223', 'C_233', 'C_243')

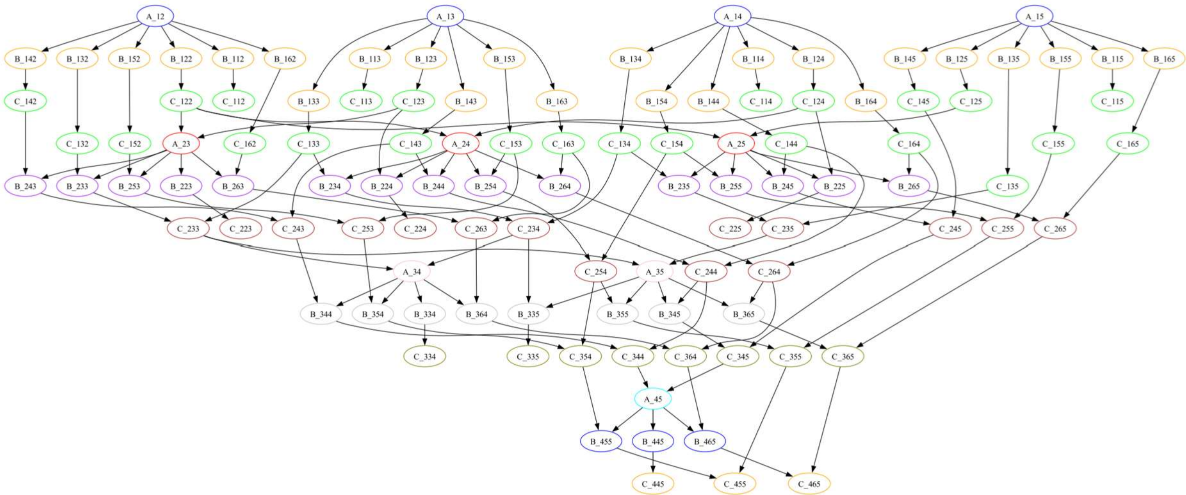


Rys. 2: Graf dla macierzy 3x3

4.3. WYNIK DZIAŁANIA DLA MACIERZY 4X4 I 5X5



Rys. 3: Graf dla macierzy 4x4



Rys. 4: Graf dla macierzy 5x5

5. WYNIK DZIAŁANIA PROGRAMU DLA CZĘŚCI IMPLEMENTACYJNEJ

Korzystając z postaci normalnej Foaty, która została wyliczona przez program dla części teoretycznej zrealizowałam równoległy algorytm eliminacji Gaussa. Operacje znajdujące się w jednej klasie są uruchamiane równolegle gdy wszystkie wątki realizujące obliczenia z klasy poprzedniej zakończą działanie.

Analizując wynik działania programu można stwierdzić, że algorytm został zaimplementowany poprawnie a zarządzanie wątkami przebiega odpowiednio.

Macierz oryginalna:

1.0	2.0	3.0
5.0	6.0	5.0

Macierz po realizacji algorytmu:

1.0	2.0	3.0
0.0	-4.0	-10.0

Wynik działania dla macierzy 2x2

Macierz oryginalna:

1.0	2.0	3.0	3.0
5.0	6.0	7.0	5.0
9.0	10.0	6.0	12.0

Macierz po realizacji algorytmu:

1.0	2.0	3.0	3.0
0.0	-4.0	-8.0	-10.0
0.0	0.0	-5.0	5.0

Wynik działania dla macierzy 3x3

Macierz oryginalna:

1.0	2.0	3.0	4.0	3.0
5.0	6.0	7.0	5.0	5.0
9.0	10.0	6.0	11.0	12.0
4.0	2.0	9.0	1.0	1.0

Macierz po realizacji algorytmu:

1.0	2.0	3.0	4.0	3.0
0.0	-4.0	-8.0	-15.0	-10.0
0.0	0.0	-5.0	5.0	5.0
0.0	0.0	0.0	16.5	13.0

Wynik działania dla macierzy 4x4

Macierz oryginalna:

2.0	1.0	2.0	3.0	4.0	3.0
1.0	5.0	6.0	7.0	5.0	5.0
6.0	9.0	10.0	6.0	11.0	12.0
12.0	4.0	2.0	9.0	1.0	5.0
10.0	16.0	7.0	3.0	4.0	1.0

Macierz po realizacji algorytmu:

2.0	1.0	2.0	3.0	4.0	3.0
0.0	4.5	5.0	5.5	3.0	3.5
0.0	0.0	-2.67	-10.33	-5.0	-1.67
0.0	0.0	0.0	23.58	-7.08	-6.58
0.0	0.0	0.0	0.0	15.28	-3.68

Wynik działania dla macierzy 5x5