Współbieżna eliminacja Gaussa

Wojciech Jasiński

24 grudnia 2022

1 Wstęp

1.1 Metoda eliminacji Gaussa

Rozwiązywanie układu równań tą metodą można rozdzielić na etap sprowadzenia do macierzy trójkątnej górnej oraz podstawianie wsteczne. Zadaniem jest uwspółbieżnić jedynie pierwszą część.

1.2 Podstawowe zadania

Jako podstawowe, niepodzielne zadania definiuję pojedyncze operacje arytmetyczne na elementach macierzy.

- A_{ik} - znalezienie mnożnika dla wiersza i, do odejmowania go od k-tego wiersza,
  $m_{ki} = M_{ki}/M_{ii}$
- B_{ijk} - pomnożenie j-tego elementu wiersza i przez mnożnik - do odejmowania od k-tego wiersza,
  $n_{ijk} = M_{ij} \cdot m_{ki}$
- C_{ijk} - odjęcie j-tego elementu wiersza i od wiersza k,
  $M_{kj} = M_{kj} - n_{ijk}$

1.3 Pseudokod

```
for i in (1..N):
    for k in (i+1..N):
        m[k, i] = M[k, i] / M[i, i]
        for j in (i+1..N+1):
            n[i, j, k] = M[i, j] * m[k, i]
            M[k, j] -= n[i, j, k]
```

Należy zwrócić uwagę, że pomijam, których wynik jest przesądzony - nie wykonuję ostatniego odejmowania od elementu pod przekątną, które miałoby go wyzerować. W późniejszych etapach obliczeń nie odwołuję się do tych elementów - zakładam, że ich wartość to zero.

1.4 Alfabet

$$\Sigma = \{A_{ik}, B_{ijk}, C_{ijk} : i \in 1..N-1; j \in 1..N+1; k \in 1..N\}$$

1.5 Zależności

Zadanie	Odczyt	Modyfikacja
A_{ik}	M_{ki}, M_{ii}	m_{ki}
B_{ijk}	M_{ij}, m_{ki}	n_{ijk}
C_{ijk}	M_{kj}, n_{ijk}	M_{kj}

Tabela 1: Zadania - odczyt i modyfikacja.

Łatwo zauważyć, że B_{ijk} i A_{ik} oraz C_{ijk} i B_{ijk} są zależne, gdyż odpowiednio pierwsza operacja wylicza wartość, która zaraz potem jest używana przez drugą. Możemy formalnie zapisać:

$$D_{AB} = \{(A_{ik}, B_{ijk}) : i \in 1..N-1; j \in 1..N+1; k \in 1..N\}$$
$$D_{BC} = \{(B_{ijk}, C_{ijk}) : i \in 1..N-1; j \in 1..N+1; k \in 1..N\}$$

Operacja C_{ijk} modyfikuje element M_{kj}, który odczytywać mogą operacje B, A albo inna operacja z grupy C. Dokładniej:

$$D_{CA1} = \{(C_{ijk}, A_{jk}) : i \in 1..N-1; j \in 1..N+1; k \in i+1..N\}$$
$$D_{CA2} = \{(C_{ijj}, A_{jk}) : i \in 1..N-1; j \in i+1..N-1; k \in j+1..N\}$$
$$D_{BC} = \{(C_{ijk}, B_{kjt}) : i \in 1..N-1; k \in i+1..N; j \in k..N+1; t \in k+1..N\}$$
$$D_{CC} = \{(C_{ijk}, C_{tjk}) : i \in 1..N-1; t \in i..N-1; j \in t..N+1; k \in t+1..N\}$$

W powyższych zapisach istotne są wartości indeksów. W każdym powyższym zbiorze reprezentujemy takie pary operacji, że pierwsza z nich modyfikuje wartość odczytywaną przez kolejną. Możemy teraz sumę powyższych zbiorów poszerzyć o zależności przechodnie, a następnie uzupełnić o symetryczne zależności.

Finalnie otrzymujemy:

$$D = sym\left\{\{D_{AB} \cup D_{BC} \cup D_{CA1} \cup D_{CA2} \cup D_{BC} \cup D_{CC}\}^{+}\right\} \cup I_{\Sigma}$$

2 Algorytm wyznaczający klasy Foaty

2.1 Algorytm

Dla zadanej wielkości macierzy mogę uruchomić skrypt, który wykona zwykłą eliminację gaussa, zapisując kolejność wykonywanych zadań. Wygenerowane w ten sposób słowo, mogę użyć do wyprowadzenia grafu zależności Diekerta oraz obliczenia klas Foaty.

Algorytm opierał się o:

1. Wyznaczenie słowa - sekwencyjny zapis zadań
2. Wyznaczenie relacji zależności

   Do identyfikacji zależności między operacjami użyłem funkcji, która na podstawie porównania wartości czytanych i modyfikowanych dla dwóch konkretnych zadań determinowała czy są zależne
3. Zbudowanie grafu oraz usunięcie krawędzi przechodnich
4. Wyznaczenie klas Foaty po wartościach najdłuższych ścieżek od źródeł w grafie

2.2 Przykład działania dla N=3

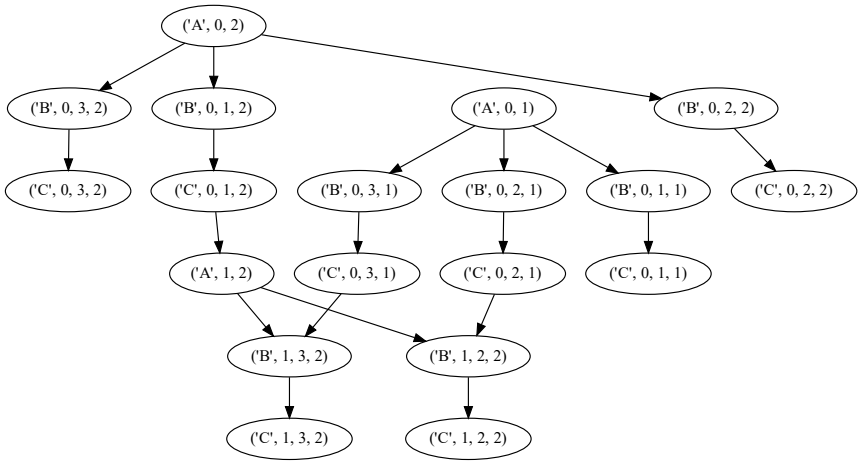
2.2.1 Sekwencyjne wywołania zadań - słowo nad alfabetem

$$A_{01}B_{011}C_{011}B_{021}C_{021}B_{031}C_{031}A_{02}B_{012}C_{012}B_{022}C_{022}B_{032}C_{032}A_{12}B_{122}C_{122}B_{132}C_{132}$$

2.2.2 Klasy Foaty

$$FNF = [A_{01}A_{02}]\,[B_{011}B_{012}B_{021}B_{022}B_{031}B_{032}]\,[C_{011}C_{012}C_{021}C_{022}C_{031}C_{032}]\,[A_{12}]\,[B_{122}B_{132}]\,[C_{122}C_{132}]$$

2.2.3 Graf Diekerta



Rysunek 1: Graf Diekerta dla N=3.

2.3 Podsumowanie

Niestety nie jest to praktyczna metoda - musimy raz wykonać obliczenia, by móc je zrównoleglić. Ponadto moja implementacja algorytmu jest zbyt wolna, by być użyteczna dla większych macierzy. Eksperymentowanie z algorytmem okazało się pomocne do wyznaczenia ogólnego rozwiązania.

3 Ogólne wyprowadzenie

3.1 Etapy obliczeń

Obliczenia można rozdzielić na etapy - niech i-tym etapem obliczeń będą zadania mające na celu odejmowanie odpowiednio przeskalowanego i-tego rzędu macierzy od pozostałych. Wszystkie zadania o indeksie pierwszym z indeksów i należą do niego.

Etap możemy rozdzielić na trzy części odpowiadające każdej z grup zadań. Każde zadanie C_{ijk} zależne jest od wyniku zadania B_{ijk}. Dla każdego j zadanie B_{ijk} zależne jest od A_{ik}. Nie uda się rozdzielić pojedynczego etapu na mniej niż trzy klasy Foaty.

Dla każdego i-tego etapu możemy znaleźć operację $C_{i,i+1,k}$ - będzie ona modyfikować element $M_{k,i+1}$, który odczytuje $A_{i+1,k}$ - jedna z pierwszych operacji w $i+1$-szym etapie. Choć pewnymi operacjami etapy mogłyby się nakładać, nie możemy pozbyć się ich sekwencyjnej natury.

Rozsądnym jest podzielić etap na trzy klasy Foaty, takie że każda odpowiada typowi zadania.

$$E_i = [A_{ik}][B_{ijk}][C_{ijk}]; j \in (i..N+1), k \in (i+1..N)$$

Zależność między zadaniem typu C z i-tego etapu, a zadaniem typu A z $i+1$-szego etapu narzuca rozdział etapów do osobnych klas.

Łącznie będziemy mieli $3*(N-1)$ klas Foaty.

$$FNF = E_1 E_2 ... E_{N-1}$$

3.2 Pseudokod Schedulera

```
for i in (1..N-1):
    scheduler.run_tasks([
        task_A(i, k) for k in (i+1..N)
    ])
    scheduler.run_tasks([
        task_B(i, j, k) for k in (i+1..N) for j in (i..N+1)
    ])
    scheduler.run_tasks([
        task_C(i, j, k) for k in (i+1..N) for j in (i..N+1)
    ])
```