Sprawozdanie 11: Programowanie dynamiczne: optymalne mnożenie wielu macierzy

Dmytro Sakharskyi - L02 - 31259 22.06.2025

Metody Programowania

Informatyka 1 rok. 2 semestr

Wstęp

Celem niniejszej pracy było zapoznanie się z problemem łańcucha mnożenia macierzy ***Matrix Chain Multiplication*** oraz implementacja dwóch algorytmów rozwiązujących ten problem: **rekurencyjnego** oraz **dynamicznego**. Problem ten polega na znalezieniu optymalnego sposobu nawiasowania iloczynu wielu macierzy w taki sposób, aby zminimalizować liczbę mnożeń skalarnych.

W ramach ćwiczenia przeprowadzono eksperymenty dla różnych zestawów danych wejściowych, analizując efektywność obu metod oraz sprawdzając, czy podane przypadki rzeczywiście dają optymalne wyniki

Zadanie 2

Kod zadania:

```cpp
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

int MatrixChainRecursive(const vector<int>& p, int i, int j) {
    if (i == j)
        return 0;

    int minCost = INT_MAX;

    for (int k = i; k < j; k++) {
        int cost = MatrixChainRecursive(p, i, k)
            + MatrixChainRecursive(p, k + 1, j)
            + p[i - 1] * p[k] * p[j];
        if (cost < minCost)
            minCost = cost;
    }

    return minCost;
}
```

MatrixChainRecursive - Rekurencyjna funkcja obliczająca minimalną liczbę mnożeń skalarnych dla danego ciągu macierzy **bez użycia dynamicznego programowania**. Działa wolno dla dużych zestawów, ale zawsze daje poprawny wynik.

```cpp
int MatrixChainDP(const vector<int>& p) {
    int n = p.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    for (int L = 2; L < n; ++L) {
        for (int i = 1; i < n - L + 1; ++i) {
            int j = i + L - 1;
            dp[i][j] = INT_MAX;
            for (int k = i; k < j; ++k) {
                int cost = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (cost < dp[i][j])
                    dp[i][j] = cost;
            }
        }
    }

    return dp[1][n - 1];
}
```

MatrixChainDP - Funkcja wykorzystująca programowanie dynamiczne (bottom-up) do optymalnego rozwiązania problemu. Tworzy tablicę dp[][], w której

przechowywane są minimalne koszty mnożenia podciągów macierzy.

```cpp
int main() {
    vector<int> p = { 10, 100, 5, 50 };

    int n = p.size() - 1;

    cout << "Liczba macierzy: " << n << "\n";
    cout << "Rozmiary macierzy:\n";
    for (int i = 0; i < n; ++i)
        cout << "  A" << (i + 1) << ": " << p[i] << "x" << p[i + 1] << "\n";

    int costRec = MatrixChainRecursive(p, 1, n);
    int costDP = MatrixChainDP(p);

    cout << "\n--- Wyniki ---\n";
    cout << "Rekurencyjnie: " << costRec << " mnozen skalarnych\n";
    cout << "Dynamicznie:   " << costDP << " mnozen skalarnych\n";

    return 0;
}
```

main():

- Ustawia dane wejściowe (rozmiary macierzy),
- Wyświetla liczbę macierzy i ich wymiary,
- Wywołuje oba algorytmy,
- Porównuje ich wyniki — liczbę mnożeń skalarnych.

Wynik działania:

```
Liczba macierzy: 3
Rozmiary macierzy:
  A1: 10x100
  A2: 100x5
  A3: 5x50

--- Wyniki ---
Rekurencyjnie: 7500 mnozen skalarnych
Dynamicznie:   7500 mnozen skalarnych
```

Wynik dla podanego ciągu macierzy (A1: 10x100, A2: 100x5, A3: 5x50):

Zarówno algorytm rekurencyjny, jak i dynamiczny zwróciły ten sam wynik:
7500 mnożeń skalarnych, co potwierdza poprawność działania obu metod.

Zadanie 3

W zadaniu sprawdzono przypadek z punktu „2. Przykład mnożenia ciągu macierzy", gdzie podano macierze o wymiarach:
A1: 10×100, A2: 100×5, A3: 5×50

Zarówno algorytm rekurencyjny, jak i algorytm dynamiczny zwróciły taki sam wynik:
7500 mnożeń skalarnych.

Potwierdza to, że podany w przykładzie wynik jest **poprawny i rzeczywiście optymalny**.

Zadanie 4

```
--- Przyklad 1: 7 macierzy ---
Rozmiary: 41 93 95 37 29 77 12 30
Minimalna liczba mnozen: 248388

--- Przyklad 2: 15 macierzy ---
Rozmiary: 75 71 16 100 52 71 20 31 85 50 51 95 100 94 100 88
Minimalna liczba mnozen: 1187792
```

W celu dokładniejszej analizy algorytmu dynamicznego dla problemu łańcucha mnożenia macierzy, przeprowadzono dwa dodatkowe testy:

- **Przypadek 1** obejmował 7 macierzy o wymiarach: 41 93 95 37 29 77 12 30.
  Minimalna liczba mnożeń skalarnych wyniosła: **248 388**.
- **Przypadek 2** zawierał aż 15 macierzy o wymiarach:
  75 71 16 100 52 71 20 31 85 50 51 95 100 94 100 88.
  W tym przypadku minimalna liczba mnożeń skalarnych wyniosła: **1 187 792**.

Na podstawie powyższych przykładów można zauważyć, że wraz ze wzrostem liczby macierzy i ich rozmiarów, złożoność obliczeniowa problemu znacznie rośnie. Algorytm dynamiczny pozwala jednak znaleźć optymalne rozwiązanie w rozsądnym czasie, nawet dla tak dużych danych.

Wnioski

Zarówno podejście rekurencyjne, jak i dynamiczne prowadzą do poprawnych wyników, jednak wersja dynamiczna jest znacznie bardziej wydajna dla większych danych.

Liczba możliwych nawiasowań **rośnie** wykładniczo wraz ze wzrostem liczby macierzy, co czyni rozwiązanie rekurencyjne niepraktycznym w przypadku większych problemów.

Weryfikacja konkretnego przykładu z treści zadania wykazała, że podane rozwiązanie (7500 mnożeń skalarnych) jest rzeczywiście optymalne.

Przeprowadzone dodatkowe testy dla **7** i **15** macierzy potwierdziły **skuteczność** algorytmu dynamicznego oraz uwidoczniły skalowanie się problemu.