Programowanie dynamiczne

Łukasz Konieczny | LK4 | Lab 8

1. Omówienie programowania dynamicznego

Programowanie dynamiczne to technika rozwiązywania problemów, w oparciu wyznaczeniu rozwiązania dla mniejszego zestawu danych, zapamiętaniu go oraz wykorzystaniu przy wyznaczaniu powiększonego zestawu danych, do momentu rozwiązania zadania dla całości danych.

Schemat rozwiązywania zadań przy pomocy PD:

- Rozwiązujemy problem dla jednego elementu, zapamiętujemy wynik
- Dodajemy kolejny element do problemu
- Rozwiązujemy zadanie dla powiększonego problemu, w oparciu o poprzednie rozwiązanie
- Kontynuujemy poprzednie kroki aż do wyznaczenia rozwiązania dla całości problemu

W odróżnieniu od metody dziel i zwyciężaj, pod problemy nie są rozłączne i cechuje je wartość optymalnej podstruktury, czyli można je rozwiązać przy pomocy algorytmu a ich optymalne rozwiązania to funkcja optymalnych rozwiązań pod problemów.

Programowanie dynamiczne ma wiele zalet:

- Problemy o strukturze rekurencyjnej pod problemów można tą metodą rozwiązać w czasie wielomianowym, bo nie musimy wielokrotnie wyznaczać rozwiązań tych samych pod problemów
- Rozwiązuje ona od razu wszystkie pod problemy danego problemu i przechowuje ich wyniki w tablicy o rozmiarze wielomianowym
- Po wyznaczeniu rozwiązań wszystkich pod problemów i ich zapisaniu, czas rozwiązywania głównego problemu zazwyczaj jest liniowy
- Pozwala ono na wyznaczenie dokładnego rozwiązania problemu, który nie posiada algorytmu wielomianowego (np. problem plecakowy)

Nie jest ono jednak pozbawione wad:

- Problem musi być sformułowany rekurencyjnie
- Musimy dowieść, że problem posiada optymalną podstrukturę
- Tablica przechowująca wyniki pod problemów zajmuje dodatkową pamięć

Programowanie dynamiczne bywa użyteczne w rozwiązywaniu wielu problemów, oto przykładowe z nich:

- Wyznaczanie n-tego wyrazu ciągu Fibonacciego
- Rozwiązywanie problemu plecakowego
- Wyznaczanie wartości silni
- Wyznaczanie współczynnika dwumiennego
- Problem podziału zbioru

2. Problem plecakowy

Problem plecakowy polega na wyznaczeniu takiego ułożenia w plecaku przedmiotów (każdy o określonej wartości oraz wadze), aby nie przekroczyć pojemności (maksymalnej wagi) „plecaka”.

Problem plecakowy ma kilka wariantów:

- Dyskretny
  - Binarny, w którym każdy przedmiot występuje tylko raz
  - Ograniczony, w którym liczba przedmiotów każdego typu jest określona
  - Nieograniczony, w którym każdy element może wystąpić dowolną ilość razy
  - Wielowymiarowy, w którym plecak i przedmioty mają co najmniej dwa wymiary
- Ciągły, w którym nie ma wymogu, aby przedmiot zapakować w całości, można „ukroić” jakąś jego część

Podczas zajęć zajmowaliśmy się tylko problemem dyskretnym binarnym, dlatego skupię się na opisaniu tylko jego.

W rozwiązaniu dynamicznym tego problemu, skorzystamy z macierzy kosztów. Będzie ona przechowywać informacje o najwyższej wartości plecaka, dla danego pod problemu.

Pod problemem będzie plecak o mniejszej pojemności, oraz zmniejszony zestaw przedmiotów. Dzięki temu wyznaczymy maksymalne ułożenie elementów w plecaku dla każdej pojemności i każdego zestawu elementów, na bazie czego zbudujemy ostateczny wynik.

Aby to osiągnąć, skorzystamy z funkcji rekurencyjnej Bellmana:

Funkcja rekurencyjna Bellmana

$$V[0,j] = 0$$ ← Dodatkowy, zerowy wiersz w macierzy PD (wyzerowany).

$$V[i,0] = 0$$ ← Dodatkowa, zerowa kolumna w macierzy PD (wyzerowana).

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \leq j \end{cases}$$

Jeśli *i*-ty element nie mieści się w plecaku, weź rozwiązanie optymalne obliczone, gdy tego elementu nie było. $x_i = 0$

Jeśli *i*-ty element mieści się w plecaku, weź maksimum z wartości:
- Plecaka bez *i*-tego elementu ($x_i = 0$)
- Plecaka z *i*-tym elementem ($x_i = 1$). Wówczas w plecaku musi być miejsce na ten element (zrób miejsce: pomniejsz *j* - aktualny rozmiar plecaka o wagę *i*-tego elementu i do rozwiązania optymalnego dodaj wartość *i*-tego elementu).

Gdzie w_i oznacza wagę elementu „i”, a p_i jego cenę.

Ostatnia komórka w macierzy będzie zawierała rozwiązanie głównego problemu:

Dane wejściowe

c = 8 Pojemność plecaka
n = 4 Ilość przedmiotów

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

W = waga przedmiotu
P = waga przedmiotu

Macierz programowania dynamicznego V

Pojemności plecaka

ID Przedmiotu

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

To jest rozwiązanie optymalne:
maksymalna wartość funkcji celu f*
(max. suma wartości elementów w plecaku).

Pierwszy wiersz macierzy kosztów na starcie wypełniamy zerami, aby nie musieć korzystać z osobnej pętli dla pierwszego z przedmiotów, tylko wykorzystać dla wszystkich funkcję rekurencyjną Bellmana w postaci określonej wyżej.

Aby odczytać które przedmioty zostały zapakowane do plecaka, ustawiamy się w ostatniej komórce i porównujemy jej wartość z komórką bezpośrednio wyżej. Jeśli komórka wyżej ma mniejszą wartość, oznacza to że nasz element jest w plecaku.

Jeśli tak jest, przesuwamy się o wiersz w górę, oraz o w_i kolumn w lewo i dokonujemy tego samego porównania.

Jeśli nie, przesuwamy się o wiersz w górę i wykonujemy to samo porównanie.

Czynność powtarzamy, aż nie dotrzemy do wiersza „0”

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

- Idziemy do V[*i*-1,*j*-w_i], na podstawie której wyznaczyliśmy V[*i*,*j*].
- Czy x_3 jest w rozwiązaniu? V[3,4] = V[2,4], więc x_3 nie jest w rozwiązaniu. Przesuwamy się o wiersz w górę, sprawdzamy czy x_2 jest w rozwiązaniu.

Algorytm cechuje złożoność pseudowielomianowa O(n * c)

3. Omówienie implementacji w C++

```cpp
struct Przedmiot{
    int cena;
    int waga;
};

void plecakuj(Przedmiot* przedmioty, int n, int c){
    // Macierz kosztów o wierszach dla danego elementu i kolumnach dla danej pojemności
    int** mPD = new int*[n + 1];
    for(int i = 0; i <= n; i++){
        mPD[i] = new int[c + 1];
    }

    // Dzięki użyciu dodatkowego wiersza wypełnionego zerami, możemy pozbyć się osobnej pętli dla piewrszego przedmiotu
    for(int i = 0; i <= c; i++){
        mPD[0][i] = 0;
    }

    for(int i = 1; i <= n; i++){
        for(int j = 0; j <= c; j++){
            if(przedmioty[i - 1].waga > j){
                mPD[i][j] = mPD[i-1][j];
            }
            else{
                mPD[i][j] = max(mPD[i-1][j], mPD[i - 1][j - przedmioty[i - 1].waga] + przedmioty[i - 1].cena);
            }
        }
    }
    cout << "Uzyskana tabela kosztow: " << endl;;
    for(int i = 0; i <= n; i++){
        for(int j = 0; j <= c; j++){
            cout << mPD[i][j] << "\t";
        }
        cout << endl;
    }

    cout << endl << endl;
    cout << "Maksymalna uzyskana wartosc plecaka: " << mPD[n][c] << endl;
    cout << "Elementy znajdujace sie w plecaku: ";

    int i = n, j = c;
    while (i != 0){
        if(mPD[i][j] > mPD[i - 1][j]){
            cout << i << ", ";
            j -= przedmioty[i - 1].waga;
        }
        i--;
    }
}
```

Aby ułatwić sobie zarządzanie przedmiotami, do przechowywania ich wartości oraz wag wykorzystuję strukturę „Przemiot”.

Do rozwiązania problemu plecakowego służy funkcja „plecakuj”, która przyjmuje tablicę jednowymiarową przedmiotów (przedmioty), jej wielkość (n) oraz maksymalną pojemność plecaka (c).

Funkcja na początku tworzy macierz kosztów, o n + 1 wierszach (musimy zostawić miejsce na w/w wiersz „0”), oraz c + 1 kolumnach (chcemy rozważyć wszystkie pojemności plecaka, od 0 do c włącznie).

Następnie wiersz „0” wypełniany jest zerami, zgodnie z założeniem powyżej.

Kolejna pętla jest główną częścią funkcji:

Rozpoczynając od wiersza „1” (czyli przedmiotu „0” w tablicy „przedmioty”, wiersz „0” nie odnosi się do żadnego przedmiotu więc musimy przesunąć indeksowanie) sprawdza, czy dany przedmiot zmieści się w danej pojemności plecaka („j”).

Jeśli nie, na to miejsce w macierzy kosztów wstawiana jest wartość odpowiadającej kolumny w wierszu wyżej (jest to poprzednie maksymalne upakowanie plecaka dla danej pojemności).

Jeśli się zmieści, sprawdzamy, czy większą wartość będzie miał plecak bez tego przedmiotu, czy z dołożonym tym przedmiotem (bierzemy maksimum z wartości plecaka bez niego, oraz z wartości plecaka o pojemności pomniejszonej o rozmiar przedmiotu [cofamy się w tabeli do maksymalnego upakowania plecaka o takiej właśnie pojemności] zwiększonej o wartość tego przedmiotu)

Wstawiamy do macierzy kosztów odpowiednią wartość.

Proces powtarzamy dla każdego przedmiotu, dla wszystkich możliwych pojemności plecaka.

Na koniec funkcja wypisuje w konsoli uzyskaną tabelę kosztów, jej ostatnią komórkę (czyli największą możliwą wartość plecaka) oraz numery przedmiotów (indeksowane od 1) znajdujących się w plecaku.

4. Podsumowanie

Podczas zajęć zapoznaliśmy się z ideą programowania dynamicznego, jego wadami oraz zaletami a także przykładami użycia, oraz zaimplementowaliśmy jeden z ważniejszych z nich, jakim jest algorytm rozwiązujący problem plecakowy

Źródła

- Materiały z wykładów – Zbigniew Kokosiński
- Artykuł o programowaniu dynamicznym w serwisie Wikipedia
- Prezentacja na temat problemu plecakowego, Politechnika Poznańska