Politechnika Krakowska

Wydział Inżynierii Elektrycznej i Komputerowej

TEMAT PROJEKTU:

Programowanie Dynamiczne

Przedmiot:

Komputerowe Wspomaganie Decyzji

Prowadzący:

mgr inż. Kazimierz Kiełkowicz

Wykonali:

Paweł Irzyk

Hubert Kopeć

1.Wprowadzenie teoretyczne

Programowanie dynamiczne jest metodologią (podobnie jak np. dziel i zwyciężaj), stosowaną przeważnie do rozwiązywania zagadnień optymalizacyjnych. Jest alternatywą dla niektórych zagadnień rozwiązywanych za pomocą algorytmów zachłannych. Wynalazcą techniki jest amerykański matematyk Richard Bellman.

Programowanie dynamiczne opiera się na podziale rozwiązywanego problemu na podproblemy względem kilku parametrów. W odróżnieniu od techniki dziel i zwyciężaj podproblemy w programowaniu dynamicznym nie są rozłączne, ale musi je cechować własność optymalnej podstruktury. Zagadnienia odpowiednie dla programowania dynamicznego cechuje również to, że zastosowanie do nich metody siłowej (ang. *brute force*) prowadzi do ponadwielomianowej liczby rozwiązań podproblemów, podczas gdy sama liczba różnych podproblemów jest wielomianowa.

Zazwyczaj jednym z parametrów definiujących podproblemy jest liczba elementów znajdujących się w rozpatrywanym problemie, drugim jest pewna wartość liczbowa, zmieniająca się w zakresie od 0 do największej stałej występującej w problemie. Możliwe są jednak bardziej skomplikowane dobory parametrów, a także większa ich liczba. Ponieważ jednak uzyskiwany algorytm zazwyczaj wymaga pamięci (i czasu) proporcjonalnego do iloczynu maksymalnych wartości wszystkich parametrów, stosowanie większej ilości parametrów niż 3-4 rzadko bywa praktyczne.

Klucz do zaprojektowania algorytmu tą techniką leży w znalezieniu równania rekurencyjnego opisującego optymalną wartość funkcji celu dla danego problemu jako funkcji optymalnych wartości funkcji celu dla podproblemów o mniejszych rozmiarach. Programowanie dynamiczne znajduje optymalną wartość funkcji celu dla całego zagadnienia, rozwiązując podproblemy od najmniejszego do największego i zapisując optymalne wartości w tablicy. Pozwala to zastąpić wywołania rekurencyjne odwołaniami do odpowiednich komórek wspomnianej tablicy i gwarantuje, że każdy podproblem jest rozwiązywany tylko raz. Rozwiązanie ostatniego z rozpatrywanych podproblemów jest na ogół wartością rozwiązania zadanego zagadnienia.

Niejednokrotnie stosowanie techniki programowania dynamicznego daje w rezultacie algorytm pseudowielomianowy. Programowanie dynamiczne jest jedną z bardziej skutecznych technik rozwiązywania problemów NP-trudnych. Niejednokrotnie może być z sukcesem stosowana do względnie dużych przypadków problemów wejściowych, o ile stałe występujące w problemie są stosunkowo nieduże. Na przykład, w przypadku dyskretnego zagadnienia plecakowego jako parametry dynamiczne w metodzie programowania dynamicznego należy przyjąć rozmiar kolejno rozpatrywanych podzbiorów elementów oraz rozmiar plecaka, zmieniający się od 0 do wartości B danej w problemie.

Programowanie dynamiczne może być również wykorzystywane jako alternatywna metoda rozwiązywania problemów zaliczanych do klasy P, o ile złożoność algorytmu wielomianowego nie jest satysfakcjonująca, a w praktyce, nawet dla dużych instancji problemu, wartości liczbowe występujące w problemie są niewielkie.

2.Przykłady

A. Ciąg Fibonacci’ego

Jednym z najbardziej powszechnych a zarazem najłatwiejszych przykładów programowania dynamicznego jest obliczanie liczby Fibonacci’ego.

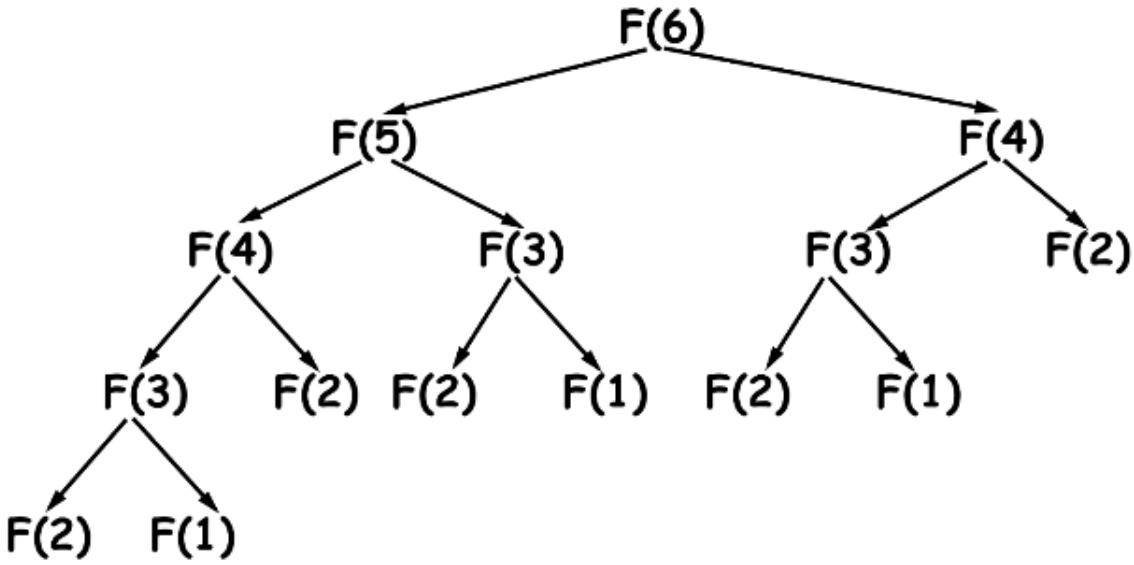
Liczby Fibonacci’ego to takie, które podążają za sekwencją:

- F(1) = 1
- F(2) = 1
- F(n) = F(n-1) + F(n-2).

Do obliczenia F(n) dla podanego n:

Jakie są podproblemy?

Znalezienie F(i) dla liczby dodatniej i mniejszej od n. Na przykład dla F(6), rozwiązanie podproblemów prezentuje grafika poniżej.



Czym jest nakładanie się podproblemów?

Jak ukazuje rysunek powyżej, są podproblemy będące obliczane wiele razy. Poprzez przechowywanie wyników, sprawiamy, że rozwiązujemy ten sam problem drugi raz bez wysiłku.

Możemy rozróżnić dwa sposoby rozwiązywania podproblemów:

- Podejście góra-dół – zaczynamy z orginalnym problemem (w tym przypadku F(n)) i rekurencyjnie rozwiązujemy mniejszy i mniejszy przypadek (F(i)) dopóki nie mamy wszystkich składników rozwiązania głównego problemu
- Podejście dół-góra – zaczynamy w z podstawowymi przypadkami (tutaj F(1) i F(2)) i rozwiązujemy coraz wyższe przypadki.

Ciąg Fibonacci'ego w Pythonie:

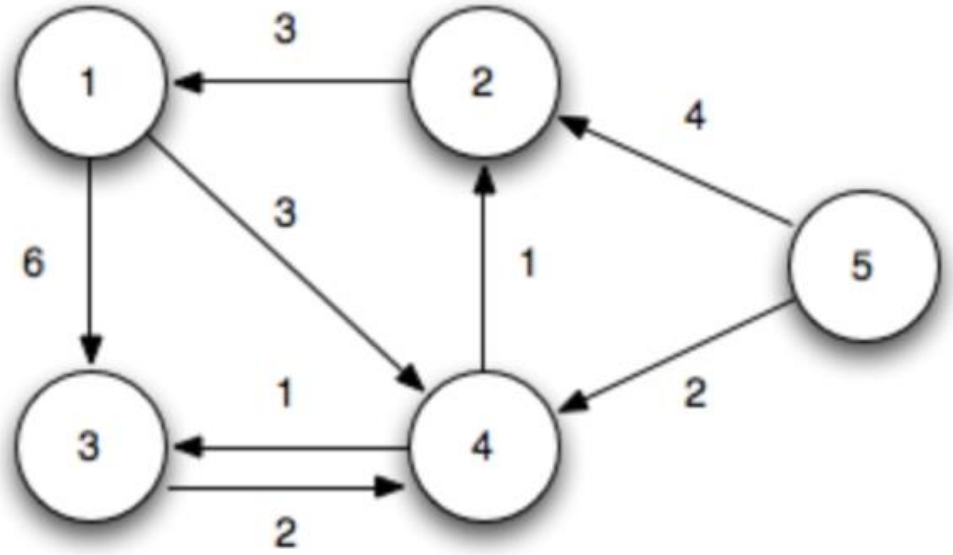
```
# Programowanie dynamiczne wstępujące, wersja z listą.
def fibonacci(n):
   F = [0] + n * [1]   # trzymamy wszystkie wartości
   for i in range(2, n+1):
      F[i] = F[i-1] + F[i-2]
   return F[n]
```

```
# Programowanie dynamiczne wstępujące, wersja ze słownikiem.
def fibonacci(n):
   F = {0:0, 1:1}   # trzymamy wszystkie wartości
   for i in range(2, n+1):
      F[i] = F[i-1] + F[i-2]
   return F[n]
```

```
# Programowanie dynamiczne zstępujące.
FIBONACCI = {0:0, 1:1}   # globalny słownik

def fibonacci(n):
   global FIBONACCI
   if n not in FIBONACCI:
      FIBONACCI[n] = fibonacci(n-1) + fibonacci(n-2)
  return FIBONACCI[n]
```

B. Znalezienie najkrótszej ścieżki w ważonym, ukierunkowanym, acyklicznym grafie



Dla powyższego grafu, począwszy od wierzchołka 1, jakie są najkrótsze ścieżki (w których sumowane masy krawędzi są minimalne) do wierzchołków 2, 3, 4 i 5?

Zanim przejdziemy do programowania dynamicznego, zaprezentujmy ten graf w tablicy ([wierzchołek_początkowy, wierzchołek_końcowy, waga]):

```
krawedzie = [
[1, 4, 3], [1, 3, 6],
[2, 1, 3],
[3, 4, 2],
[4, 3, 1], [4, 2, 1],
[5, 2, 4], [5, 4, 2]
]
```

Jakie są podproblemy?
Dla liczby nieujemnej i, podając że jakakolwiek ścieżka zawiera co najwyżej krawędzie, jaka jest najkrótsza ścieżka od początku wierzchołka do innych wierzchołków?

Jak rozwiązać podproblemy?
Zacznijmy od podstawowego przypadku, czyli i = 0, dystans w tym przypadku do wszystkich wierzchołków prócz startowego jest nieskończona, a dystans do startowego wierzchołka wynosi 0.
Dla i od 1 do liczby_wierzcholkow-1 (najdłuższa z najkrótszych ścieżek do jakiegokolwiek wierzchołka zawiera co najwyżej tyle krawędzi, zakładając że nie ma ujemnej wagi cyklu), przechodzimy przez wszystkie krawędzie:
Dla każdej krawędzi, obliczamy nowy dystans krawedz[2] + dystans_do_wierzcholka_krawedzi[0], jeśli nowy dystans jest mniejszy niż dystans_do_wierzcholka_krawedzi[1], aktualizujemy dystans_do_wirzcholka_krawedzi[1] nowym dystansem.

C. Problem wydawania reszty

Sformułowanie problemu:

Dany jest posortowany rosnąco ciąg nominałów A=(c1, c2, …, cn) oraz kwota do wydania r. Należy wyznaczyć takie nieujemne współczynniki k1, k2, …, kn, że k1c1 + k2cc + … + kncn = r, a suma k1 + k2 + … + kn jest jak najmniejsza.

Zakładamy, że wszystkie nominały i kwota r są liczbami naturalnymi, a nominał c1=1. Przyjmujemy również że wartości k nie mają górnych ograniczeń.

Opis algorytmu:

Ten algorytm polega na rozstrzyganiu, czy do wydania danej kwoty opłacalne jest użycie najwyższego dostępnego nominału, czy też nie. Oznaczmy jako oi,j optymalną (najmniejszą z możliwych) liczbę monet potrzebnych do wyznaczenia kwoty i za pomocą nominałów (c1, c2, …, cj). Aby wyznaczyć wartość oi,j musimy wiedzieć, co jest większe: oi-aj,j+1 (jedna moneta najwyższego dostępnego nominału + liczba monet potrzebna do wydania pozostałej kwoty) czy oi,j-1 (skorzystanie z rozwiązania dla mniejszego zbioru nominałów). Rozwijając to zagadnienie rekurencyjnie dojdziemy w końcu do przypadków oczywistych (np. wydanie kwoty tylko za pomocą nominału 1 lub wydanie kwoty równej jednemu z nominałów). Formalnie możemy to zapisać jako:

- oi,j =
- i dla j=1
- 1 dla i=cj
- oi,j-1 dla i<cj, j>1
- min(oi,j-1, oi-cj,j+1) dla i>cj, j>1

Aby nie wyznaczać wielokrotnie tej samej wartości, wyniki należy przechowywać w następującej tabeli:

$o_{1,1}$	$o_{1,2}$	…	$o_{1,n}$
$o_{2,1}$	$o_{2,2}$	…	$o_{2,n}$
…	…		…
$o_{r,1}$	$o_{r,2}$	…	$o_{r,n}$

Warto zauważyć, że w ten sposób wyznaczamy jedynie łączną liczbę potrzebnych monet, a nie poszczególne współczynniki ki. Dlatego też w każdej komórce tabeli oprócz wartości oi,j należy zapamiętać również wartość si,j oznaczającą, od którego nominału należy rozpocząć wydawanie kwoty i. Wartości te wyznacza się następująco:

- si,j =
- 1 dla j=1
- j dla i=cj lub jeśli wartości oi,j przypisaliśmy wartość oi-cj,j+1
- si,j-1, jeśli wartości oi,j przypisaliśmy wartość oi,j-1

Mając wyznaczone wartości si,j poszczególne współczynniki ki można wyznaczyć w prosty sposób. Na początku wszystkim współczynnikom ki przypisujemy wartość 0. Następnie sprawdzamy wartość sr,n i zwiększamy wartość współczynnika ksr,n o 1. W kolejnym kroku zmniejszamy wartość r o csr,n i odczytujemy kolejną wartość sr,n. Czynności powtarzamy, dopóki r>0.

Tabelę z wartościami można wypełniać rekurencyjnie, jednak częściej wypełnia się ją w sposób iteracyjny (od wartości o1,1 do or,n. Jeśli tabelę wypełniamy w sposób iteracyjny kolumnami, to możemy zaoszczędzić pamięć zapamiętując tylko jedną kolumnę (nadpisując wartości zamiast zapisywania ich w kolejnej kolumnie).

Złożoność obliczeniowa:

Tabela tworzona w trakcie wykonywania algorytmu ma n na r komórek, gdzie n jest liczbą nominałów a r kwotą do wydania. Złożoność czasowa algorytmu jest zatem O(nr). Złożoność pamięciowa to O(nr) jeśli pamiętamy wszystkie kolumny i O(r) jeśli je nadpisujemy.

Przykład:

Załóżmy, że chcemy wyrazić kwotę 6 mając monety o nominałach 1, 3 i 4. Aby wyrazić tę kwotę za pomocą jak najmniejszej liczby monet, musimy wiedzieć, czy warto użyć monety o największym nominale (w tym przypadku 4), czy też nie. W naszym przypadku musimy zatem wiedzieć:

Ile monet potrzeba do wydania kwoty 6 jedynie za pomocą nominałów 1 i 3?

Ile monet potrzeba do wydania kwoty 2 za pomocą pełnej puli nominałów?

Wówczas wiemy, czy lepiej skorzystać z najwyższego dostępnego nominału, czy też wykorzystać rozwiązanie dla pomniejszonego zbioru nominałów. Aby odpowiedzieć na powyższe pytania, musimy rekurencyjnie stawiać kolejne, aż dojdziemy do przypadków oczywistych. Przygotujmy sobie zatem tabelkę, zgodnie z opisanym powyżej algorytmem. W pełni wypełniona tabela wygląda następująco:

Kwota do wydania	Zbiór nominałów		
	1	1, 3	1, 3, 4
1	1 (1)	1 (1+0)	1 (1+0+0)
2	2 (2)	2 (2+0)	2 (2+0+0)
3	3 (3)	1 (0+1)	1 (0+1+0)
4	4 (4)	2 (1+1)	1 (0+0+1)
5	5 (5)	3 (2+1)	2 (1+0+1)
6	6 (6)	2 (0+2)	2 (0+2+0)

D. Problem plecakowy

Sformułowanie problemu:

Dyskretny problem plecakowy jest jednym z najczęściej poruszanych problemów optymalizacyjnych. Nazwa zagadnienia pochodzi od maksymalizacyjnego problemu wyboru przedmiotów, tak by ich sumaryczna wartość była jak największa i jednocześnie mieściły się w plecaku. Przy podanym zbiorze elementów o podanej wadze i wartości, należy wybrać taki podzbiór by suma wartości była możliwie jak największa, a suma wag była nie większa od danej pojemności plecaka.

Opis algorytmu:

Tworzymy macierz V 0[..n, 0 , ..b ,] gdzie1 ≤ i ≤ n 0 i ≤ s ≤ b , element

V[i, s] przechowuje maksymalną wartość dla dowolnego podzbioru elementów {1, 2, … , i} o sumarycznym rozmiarze nie większym niż s.

Jeśli obliczymy wszystkie elementy tej macierzy to element V[n,b]będzie zawierał rozwiązanie problemu.

$$\begin{cases} V[0,s] = 0 & dla\ 0 \le s \le b \\ V[i,s] = \max(V[i\text{-}1, s], w(a_i) + V[i\text{-}1, s\text{-}s(a_i)]) & dla\ 1 \le i \le n,\ 0 \le s \le b \end{cases}$$

gdzie

V[i −1,s] oznacza najlepsze (poprzednio obliczone i zapamiętane w tabeli) rozwiązanie dla rozważanej aktualnie pojemności plecaka s oraz podzbioru elementów {1, … i-1}

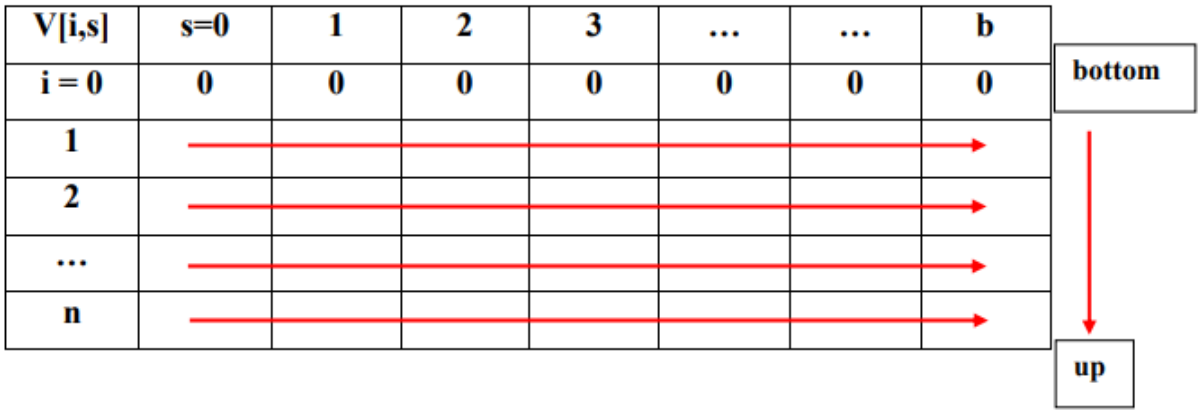
V[i-1 ,s-s(a_i)] oznacza najlepsze (poprzednio obliczone i zapamiętane w tabeli)

rozwiązanie dla aktualnie rozważanej pojemności plecaka s pomniejszonej o rozmiar elementu s(a_i), który próbujemy dołożyć do plecaka, dla podzbioru elementów {1, … i-1}

Obliczenie optymalnego kosztu metodą wstępującą:

Wypełnij pierwszy wiersz zgodnie ze wzorem V[0,s] = 0 dla 0 ≤ s ≤ b(bottom)

Oblicz pozostałe elementy macierzy używając wzoru V[i,s]= max(V[i-1, s], $w(a_i)$ + V[i-1, s-$s(a_i)$]) dla 1 ≤ i ≤ n, 0 ≤ s ≤ b Wypełniaj tabelę rząd po rzędzie.



Przykład:

Dane:

b = 10

i	1	2	3	4
$w(a_i)$	10	40	30	50
$s(a_i)$	5	4	6	3

Rozwiązanie:

V[i,s]	0	1	2	3	4	5	6	7	8	9	10
i=0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Przykładowe obliczenia dla poszczególnych elementów tabeli:

V[1,5] = max{V[0,5], 10 + V[0, 5-5]} = 10

V[2,4] = max{V[1,4], 40 + V[1, 4-4]} = 40

V[2,9] = max{V[1,9], 40 + V[1, 5]} = max{10, 40 + 10} = 50

V[3,4] = max{V[2,4], 0} = 40

V[3,6] = max{V[2,6], 30 + V[2, 6-6]} = max{40, 30} = 40

V[4,8] = max{V[3,8], 50 + V[3, 8-3]} = max{40, 50 + 40} = 90

Przedstawiona metoda nie daje informacji, który podzbiór elementów dał rozwiązanie optymalne (w tym przypadku były to elementy {2, 4}).