Problema Rucsacului - Knapsack Problem

Gabriel-Ioan PAVEL, Andrei RUSĂNESCU, and Amir FALLAH-MIRZAEI

Universitatea de Știință și Tehnologie POLITEHNICA București

Rezumat Acest raport reprezintă analiza câtorva algoritmi ce rezolvă Problema Rucsacului

1 Introducere

Problema Rucsacului reprezintă problema alocării și gestionării de resurse limitate, unde ținta este obținerea unui profit maxim.

Aplicații practice Printre domeniile în care apare Problema Rucsacului se numără: găsirea unei metode cât mai puțin risipitoare de a tăia materii prime, selecția investițiilor și a portofoliilor, selectarea activelor pentru securitizare garantată cu active etc.

2 Demonstrație NP-Hard

3 Prezentarea algoritmilor

3.1 Programare dinamică

Rezolvarea problemei prin metoda programării dinamice reprezintă construirea unei matrici ce stochează profitul maxim pentru subproblemele definite de primele **i** obiecte și o capacitate **w** a rucsacului.

```
int knapsack(const int &maxWeight, std::vector<int> &weights,
    std::vector<int> &values) {
  // numar de obiecte
  int n = weights.size();

  /* matrice - dp[i][w] == profitul maxim ce poate fi obtinut folosind
      primele i obiecte cu limita de masa w */
  std::vector<std::vector<int>> dp(n + 1, std::vector<int>(maxWeight +
      1));

  /* daca nu intra niciun obiect in calcul sau masa maxima este 0,
      atunci profitul este 0 */
  for (int i = 0; i <= n; ++i)
    dp[i][0] = 0;
  for (int j = 0; j <= maxWeight; ++j)
    dp[0][j] = 0;
```

Următoarea buclă parcurge fiecare element al matricei, unde fiecare linie reprezintă un obiect, iar fiecare coloană reprezintă o masă maximă, astfel încât dp[i][w] reprezintă profitul maxim obținut folosind primele **i** obiecte cu limita de masă **w**.

Logica este următoarea: dacă obiectul curent se încadrează în masa maximă curentă, atunci alegem să îl includem sau să îl excludem din calcul, în funcție de profitul maxim pe care îl putem obține. Mai exact, dacă alegem să îl includem, adunăm valoarea sa cu profitul maxim care se poate obține având în vedere masa folosibilă rămasă, iar, dacă alegem să îl excludem, păstrăm profitul maxim ce poate fi obținut precedent.

Contrar, dacă masa obiectului curent întrece limita curentă, acesta va fi exclus din calcul, păstrându-se profitul precedent.

```
   for (int i = 1; i <= n; ++i) {
      for (int w = 1; w <= maxWeight; ++w) {
         if (weights[i - 1] <= w)
            dp[i][w] = std::max(values[i - 1] + dp[i - 1][w - weights[i -
                1]], dp[i - 1][w]);
         else
            dp[i][w] = dp[i - 1][w];
      }
   }

   return dp[n][maxWeight];
}
```

Rezultatul căutat se află pe ultima poziție din tabel.

Analiza complexității

Complexitate temporală Primele două bucle *for* execută **n**, respectiv **maxWeight** operații de timp constant. Pentru următoarele două bucle, cea exterioară iterează prin fiecare obiect, iar cea interioară prin fiecare masă maximă posibilă. Operațiile executate în interiorul buclei interne sunt de timp constant. Astfel, complexitatea temporală a algoritmului este de **O(n * maxWeight)**.

Complexitate spațială Întrucât este folosită o matrice de dimensiuni **(n+1) * (maxWeight+1)**, complexitatea spațială este **O(n * maxWeight)**.

Avantaje și dezavantaje

Avantaje Avantajele principale sunt reprezentate de viteza de execuție, întrucât se poate obține un rezultat în timp rezonabil și evitarea problemelor ce ar fi putut apărea în situația folosirii recursivității, precum supraîncărcarea stivei.

Dezavantaje Dezavantajele principale sunt reprezentate de folosirea unei zone mari de memorie și faptul că dimensiunea matricii nu poate fi ajustată dinamic fără recalcularea valorilor.

3.2 Memoizare

Această metodă de programare dinamică se folosește de memoizare și recursivitate pentru a evita recalcularea inutilă a unor subprobleme.

Execuția începe în funcția auxiliară, **knapSackMemoHelper**, care inițializează variabilele și începe execuția algoritmului propriu-zis.

```
inline int knapSackMemoHelper(int W, std::vector<int> &weights,
    std::vector<int> &vals) {
   int n = weights.size();
   std::vector<std::vector<int>> memo(n, std::vector<int>(W + 1, -1));

   /* W - capacitatea maxima curenta, index - indicele obiectului curent
       */
   return knapSackMemo(W, weights, vals, n - 1, memo);
}
```

Funcția principală începe prin a verifica cazul opririi recursivității - s-au luat în calcul toate obiectele - ca apoi să verifice dacă profitul pentru obiectul curent și capacitatea W a fost deja calculat. În caz afirmativ, rezultatul este întors direct, fără a mai fi făcute calcule redundante.

Algoritmul verifică dacă masa obiectului curent este mai mare decât capacitatea rămasă în rucsac. Dacă da, obiectul trebuie exclus. Altfel, se calculează profitul maxim care se poate obține din includerea sau excluderea obiectului curent, similar cu metoda anterioară, și rezultatul este păstrat în matricea de memoizare.

```
int knapSackMemo(int W, std::vector<int> &weights, std::vector<int>
    &vals, int index, std::vector<std::vector<int>> &memo) {
   if (index < 0)
      return 0;

   if (memo[index][W] != -1)
      return memo[index][W];

   if (weights[index] > W)
      return memo[index][W] = knapSackMemo(W, weights, vals, index - 1,
          memo);

   return memo[index][W] = std::max(
      knapSackMemo(W, weights, vals, index - 1, memo),
      vals[index] + knapSackMemo(W - weights[index], weights, vals, index
          - 1, memo));
}
```

Analiza complexității

Complexitate temporală Fiecare subproblemă este unică, este rezolvată o singură dată și este definită de indicele unui obiect și capacitatea rămasă în rucsac într-un moment. Astfel, ajungem la un număr de **n * maxWeight** de subprobleme, iar complexitatea temporală ajunge să fie **O(n * maxWeight)**, întrucât operațiile ce trebuie executate pentru a rezolva subproblemele sunt de timp constrant.

Complexitate spațială Fiind folosită o matrice de dimensiune **n * (maxWeight + 1)**, iar stiva având o dimensiune maximă de **n**, complexitatea spațială este de aproximativ **O(n * maxWeight)**.

Avantaje și dezavantaje

Avantaje Principalele avantaje ale acestei abordări sunt reprezentate de viteza de execuție și calculul soluțiilor strict necesare, comparativ cu metoda programării dinamice, unde toate elementele matricii sunt calculate.

Dezavantaje Principalele dezavantaje sunt reprezentate de utilizarea unei zone mari de memorie și riscul unui stack overflow, atunci când **n** este foarte mare.

4 Evaluare

5 Concluzii

Bibliografie

1. Pavel, G.-I.: *Knapsack problem*, `https://en.wikipedia.org/wiki/Knapsack_problem` [Accesat la 04/01/2025].