The Knapsack Problem

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

Universitatea de Știință și Tehnologie POLITEHNICA București

Abstract. The present paper analyses several algorithms that solve the Knapsack problem, taking into account both the speed and the memory used by the program. It presents three solutions, two of which always generate the correct answer, and one that obtains a result close to the right one.

1 Introduction

The Knapsack problem derives its name from the situation faced by someone who wants to fill their limited-capacity knapsack with the most valuable items that can fit. You can choose whether to put it in the bag or not put it at all, hence the name "0/1 Knapsack Problem". The problem has been studied for more than a century, with writings dating as far back as 1897.

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.

3.3 Backtracking

Această abordare verifică fiecare combinație de obiecte posibilă în mod recursiv. Funcția care începe algoritmul este una auxiliară, având ca scop inițializarea variabilelor și apelarea funcției principale. Aici, **maxxW** reprezintă masa maximă suportată de rucsac, **maxx** profitul maxim într-un anumit moment, **size** numărul de obiecte, **ws** și **vals** copii ale vectorilor de mase, respectiv valori ale obiectelor. Funcția se finalizează prin returnarea rezultatului final.

```
inline int knapsack_back(const int &maxWeight, std::vector<int>
    &weights, std::vector<int> &values) {
  maxx = 0;
  maxxW = maxWeight;
  ws = weights;
  vals = values;
  size = weights.size();
  back(0, 0, 0);
  return maxx;
}
```

Fiecare apel al funcției principale verifică dacă obiectul curent se încadrează în masa maximă curentă a rucsac. Dacă da, actualizează profitul maxim curent, altfel încheie execuția. Dacă s-au luat toate obiectele în considerare, execuția este oprită.

În continuare, funcția iterează prin toate obiectele începând cu un **index**. Pentru fiecare obiect, adaugă masa și valoarea sa la profitul curent și masa folosită curentă și merge mai departe în recursivitate. După finalizarea recursivității, elimină valoarea și masa sa din variabilele corespunzătoare.

```
void back(int index, int currWeight, int currProfit) {
   if (currWeight <= maxxW)
      maxx = std::max(maxx, currProfit);
   else return;

   if (index == size) {
      return;
   }

   for (int i = index; i < size; ++i) {
      currProfit += vals[i];
      currWeight += ws[i];
      back(i + 1, currWeight, currProfit);
      currProfit -= vals[i];
      currWeight -= ws[i];
   }
}
```

Analiza complexității

Complexitate temporală Toate operațiile de bază executate la fiecare iterație în recursivitate sunt de timp liniar, iar, din moment ce algoritmul explorează toate combinațiile posibile de obiecte, ajungem la un număr de 2^n iterații. Astfel, complexitatea temporală a algoritmului este de $\mathbf{O(2^n)}$.

Complexitate spațială Având **n** elemente de verificat, dimensiunea stivei de execuție ajunge la maxim **n**, deci complexitatea spațială este de $\mathbf{O(n)}$.

Avantaje și dezavantaje

Avantaje Avantaje acestei abordări sunt reprezentate de simplitatea în implementarea algoritmului și dimensiunea relativ redusă a memoriei utilizate.

Dezavantaje Întrucât această metodă este una de bruteforce, durata de execuție este foarte ridicată și se realizează foarte multe calcule redundante, ineficiența sa devine un un dezavantaj masiv.

3.4 Recursivitate

Abordarea recursivă parcurge toate combinațiile posibile de obiecte fără a păstra valori intermediare.

Algoritmul începe prin a trata cazul de bază, unde fie nu mai este spațiu disponibil în rucsac, fie s-au luat toate obiectele în considerare, caz în care execuția se încheie. În continuare, verifică dacă masa obiectului curent se încadrează în masa maximă rămasă. În cazul afirmativ, obiectul nu poate fi luat în calcul, iar recursivitea continuă fără a-l include. În caz contrat, se decide dacă valoarea obiectului curent poate contribui la creșterea profitului maxim și se include sau exclude în funcție de concluzie, similar cu algortimii precedenți.

```
int knapSack(int W, std::vector<int> &weights, std::vector<int> &vals,
    int n) {
  if (n == 0 || W == 0)
    return 0;

  if (weights[n - 1] > W)
    return knapSack(W, weights, vals, n - 1);

  return std::max(knapSack(W, weights, vals, n - 1),
            vals[n - 1] + knapSack(W - weights[n - 1], weights, vals,
                n - 1));
}
```

Analiza complexității

Complexitate temporală Algoritmul explorează toate combinațiile posibile de obiecte, ajungându-se astfel la o complexitate temporală de $\mathbf{O(2^n)}$.

Complexitate spațială Cum numărul de obiecte este **n**, adâncimea maximă a stivei poate fi de **n**, deci complexitatea spațială este de $\mathbf{O(n)}$.

Avantaje și dezavantaje

Avantaje Avantajele acestei abordări sunt aproape inexistente, singurele fiind simplitatea implementării și dimensiunea relativ redusă a stivei de execuție.

Dezavantaje Timpul de execuție este cel mai dezavantajos aspect al acestui algoritm, fiind foarte lent, în special pe intrări de dimensiune ridicată.

4 Evaluare

5 Concluzii

References

1. Pavel, G.-I.: *Knapsack problem*, https://en.wikipedia.org/wiki/Knapsack_problem [Accesat la 04/01/2025].
2. LNCS Homepage, http://www.springer.com/lncs, Accesat la [04/01/2025]