Proiect Analiza Algoritmilor
Problema Rucsacului

Ivan Alexandru, Trufaș Rareș, Stîngă Cristina - 322CC

Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA București

1 Introducere

Problema rucsacului (Knapsack Problem) este una dintre cele mai cunoscute probleme de optimizare din domeniul informaticii teoretice. Apărută inițial ca o provocare matematică, problema rucsacului are aplicații practice extinse, fiind utilizată în economie, logistică, planificare și alte domenii conexe.

Problema se concentrează pe optimizarea utilizării resurselor limitate, având ca scop selectarea unui subset optim de elemente astfel încât să se maximizeze un beneficiu total, fără a depăși o anumită constrângere de capacitate.

1.1 Descrierea problemei

Problema rucsacului presupune un set de n obiecte, fiecare având o valoare v_i și o greutate g_i. Este disponibil un rucsac cu o capacitate maximă G. Obiectivul este de a determina un subset de obiecte care maximizează valoarea totală $\sum_{i \in S} v_i$, astfel încât greutatea totală $\sum_{i \in S} g_i$ să nu depășească capacitatea G.

Această problemă poate avea mai multe variante, cum ar fi:

- **Problema rucsacului 0/1**: Obiectele pot fi fie selectate integral, fie deloc.
- **Problema rucsacului fracționar**: Obiectele pot fi selectate parțial, împărțindule în fracții.
- **Problema rucsacului multidimensional**: Există mai multe constrângeri suplimentare.
- **Problema rucsacului nelimitat**: Nu există o limită superioară pentru numărul de ori în care un obiect poate fi selectat.

1.2 Exemple de aplicații practice

Problema rucsacului are aplicații vaste în diferite domenii, dintre care menționăm:

- **Logistică și transport**: Optimizarea încărcării unui vehicul de transport, având constrângeri de greutate și spațiu.
- **Investiții financiare**: Alegerea unui portofoliu de investiții care maximizează profitul în funcție de constrângerile bugetare.
- **Planificare și management de proiect**: Alocarea resurselor limitate pentru a maximiza rezultatele unui proiect.

- **Compresia datelor**: Selecția optimă a subseturilor de date pentru stocare eficientă.
- **Inteligență artificială**: Probleme de alocare a resurselor în rețele neuronale sau învățare automată.

Scopul acestui proiect este de a analiza în detaliu **problema rucsacului 0/1**, de a implementa soluții algoritmice eficiente și de a explora aplicațiile acesteia în situații reale. Vom prezenta atât metode exacte, precum programarea dinamică, cât și tehnici euristice utilizate pentru instanțele de mari dimensiuni. Astfel, acest proiect își propune să ofere o perspectivă cuprinzătoare asupra unei probleme fundamentale din matematică și informatică.

2 Demonstrație NP-Hard

Problema rucsacului 0/1 este considerată NP-completă, iar acest lucru poate fi demonstrat prin reducerea unei probleme deja cunoscute a cărei complexitate este NP-hard.

2.1 Enunțul problemei

Input: O mulțime $O = \{o_1, o_2, \ldots, o_N\}$ de obiecte, fiecare având o greutate g_i și o valoare v_i, și o capacitate maximă G a rucsacului.
Output: Se dorește găsirea unui subset $S \subseteq O$, astfel încât:

- suma greutăților $\sum_{i \in S} g_i \leq G$,
- suma valorilor $\sum_{i \in S} v_i$ să fie maximă.

2.2 Teoremă: Problema rucsacului este NP-completă

Plan de demonstrație:

1. **Este în NP:** Se poate verifica în timp polinomial dacă soluția respectă constrângerile impuse.
2. **Subset Sum $\leq_p$ Problema rucsacului:** Vom reduce problema Subset Sum la problema rucsacului, demonstrând astfel că problema rucsacului este NP-hard.

Pasul 1: Este în NP Se poate verifica într-un timp polinomial dacă suma greutăților selectate nu depășește capacitatea rucsacului și dacă suma valorilor este maximă. Aceste verificări sunt operații care pot fi efectuate într-un timp polinomial, astfel că problema rucsacului 0/1 este în NP.

Pas 2: Alegem Subset Sum

- **Input:** $M = \{x_1, x_2, \ldots, x_N\}$ numere pozitive și o valoare țintă Sum.
- **Întrebare:** Există un subset $S \subseteq M$ astfel încât $\sum_{x \in S'} x = \text{Sum}$?

- Fie (M, Sum) un input pentru Subset Sum.
- Construim instanța (O, G) pentru Problema Rucsac.
- Pentru fiecare $x_i \in M$, construim un obiect $o_i = (x_i, x_i)$ în O, unde $g_i = v_i = x_i$.
- Setăm capacitatea rucsacului $G = \text{Sum}$.

Corectitudinea reducerii:

$\Rightarrow$ Dacă $S \subseteq M$ este o soluție pentru Subset Sum:
1. Dacă S este un subset al lui M a.î. $\sum_{x \in S} x = \text{Sum}$:
   - S corespunde unui subset $S' = \{(x_i, x_i) \mid x_i \in S\} \subseteq O$ în Problema Rucsacului.

- Greutatea totală $\sum_{i \in S'} g_i = \sum_{x \in S} x = \text{Sum}$.
- Valoarea totală $\sum_{i \in S'} v_i = \sum_{x \in S} x = \text{Sum}$.

2. Subsetul S' respectă constrângerea privind greutatea $\sum_{i \in S'} g_i \leq G$, iar suma valorilor $\sum_{i \in S'} v_i = Sum(\text{maximă}) \Rightarrow$ soluție pentru Problema Rucsacului.

$\Leftarrow$ Dacă $S' \subseteq O$ este o soluție optimă pentru Problema Rucsacului:

1. Presupunem următoarele pentru subsetul S':
   - Greutatea totală $\sum_{i \in S'} g_i <= G = \text{Sum}$.
   - Valoarea totală $\sum_{i \in S'} v_i$ este maximă.
2. Pentru că $g_i = v_i$ pentru fiecare obiect $i \Rightarrow \sum_{i \in S'} g_i = \sum_{i \in S'} v_i$.
3. Subsetul S' corespunde unui subset $S \subseteq M$ din problema Subset Sum unde $\sum_{x \in S} x = \text{Sum}$.

Complexitatea reducerii:

- Construiesc mulțimea de obiecte O unde fiecare $x_i \in M$ este transformat într-un obiect o_i care are $g_i = v_i = x_i \rightarrow O(|M|)$.
- Setez capacitatea rucsacului $(G) \rightarrow O(1)$.
- Reducerea completă are complexitatea $O(|M|) \rightarrow$ POLINOMIALĂ.

3 Prezentarea Algoritmului

3.1 Modul de funcționare

Acest proiect implementează trei metode principale pentru rezolvarea problemei rucsacului:

Programarea dinamică Funcționează prin construcția unei matrice C unde $C[i][j]$ reprezintă valoarea maximă care poate fi obținută folosind primele i obiecte și având o capacitate totală de j.
Se parcurg obiectele și greutățile posibile iterativ, calculând valoarea optimă și decizia de a include sau nu fiecare obiect.
Rezultatul final este în $C[n][G]$, unde n este numărul total de obiecte și G este capacitatea maximă a rucsacului.

Listing 1.1. Metoda programării dinamice

```
def knapsack_dynamic(n, G, a):
    C = [[0]*(G+1) for _ in range(n+1)]

    for i in range(1, n+1):
        for j in range(1, G+1):
            if a[i-1].g <= j and
               a[i-1].c+C[i-1][j-a[i-1].g] > C[i-1][j]:
                C[i][j] = a[i-1].c+C[i-1][j-a[i-1].g]
            else:
                C[i][j] = C[i-1][j]

    return C[n][G]  # Return the optimal value
```

Backtracking Explorează toate combinațiile posibile de obiecte, adăugând sau excluzând fiecare obiect din soluția curentă. Utilizează recursivitatea pentru a ține evidența greutății rămase și a valorii curente. Stabilește cea mai bună valoare (optimizare globală) dintre toate combinațiile posibile.

Listing 1.2. Metoda backtracking

```
def knapsack_backtracking(n, W, items):
    def backtrack(i, remaining_weight, current_value):
        nonlocal best_value

        if i == n:
            best_value = max(best_value, current_value)
            return

        if items[i].g <= remaining_weight:
            backtrack(i + 1, remaining_weight - items[i].g,
    current_value + items[i].c)

        backtrack(i + 1, remaining_weight, current_value)

    best_value = 0
    backtrack(0, W, 0)
    return best_value
```

Greedy Sortează obiectele în funcție de raportul valoare/greutate într-o ordine descrescătoare. Adaugă obiectele în rucsac până când greutatea totală depășește capacitatea, și adaugă fracțiunea corespunzătoare din ultimul obiect. Returnează valoarea totală acumulată.

Listing 1.3. Greedy solution for the knapsack problem

```
def fractional_knapsack(n, capacity, items):
    items.sort(key=lambda x: x.c / x.g, reverse=True)

    total_value = 0.0
    for item in items:
        if capacity >= item.g:
            total_value += item.c
            capacity -= item.g
        else:
            break

    return total_value
```

3.2 Analiza complexității

Programarea dinamică

- **Complexitatea temporală**: $O(n \cdot G)$, unde n este numărul de obiecte și G este capacitatea rucsacului.
- **Complexitatea spațială**: $O(n \cdot G)$ pentru stocarea matricei C.
- Metoda garantează o soluție optimă, dar devine ineficientă pentru valori mari ale lui G datorită cerințelor ridicate de spațiu și timp.

Backtracking

- **Complexitatea temporală**: $O(2^n)$, deoarece explorează toate combinațiile posibile de selecție a obiectelor.
- **Complexitatea spațială**: $O(n)$, pentru stiva apelurilor recursive.
- Această metodă oferă o soluție optimă, dar este practic ineficientă pentru n mare.

Greedy

- **Complexitatea temporală**: $O(n \log n)$, datorită sortării obiectelor după raportul valoare/greutate.
- **Complexitatea spațială**: $O(1)$, deoarece nu necesită stocarea unor structuri suplimentare semnificative.
- Metoda nu garantează o soluție optimă pentru problema rucsacului $0/1$, dar funcționează bine pentru problema fracționară.

4 Evaluare

Această secțiune prezintă metodologia utilizată pentru testarea algoritmului $0/1$ Knapsack, rezultatele evaluărilor și interpretarea valorilor obținute.

4.1 Construirea testelor

Pentru a evalua performanța algoritmului, am generat un set de teste utilizând un script Python dedicat. Scriptul generează două categorii de cazuri de test:

- **Categoria 1:** Teste cu număr variabil de obiecte (n) și capacitate fixă (G).
- **Categoria 2:** Teste cu capacitate variabilă (G) și număr fix de obiecte (n).

Scriptul are următoarele caracteristici:

- Numărul total de teste generate este specificat de utilizator (`num_tests`).
- Fiecare obiect este reprezentat printr-un tuplu $(weight, value)$, unde greutatea și valoarea sunt generate aleator între limite specificate (`max_weight` și `max_value`).

- Capacitatea rucsacului (G) și numărul de obiecte (n) sunt generate în funcție de categoria testului.
- Testele sunt salvate în fișiere text, fiecare conținând pe primul rând numărul de obiecte și capacitatea rucsacului, iar pe următoarele rânduri greutatea și valoarea fiecărui obiect.

Codul complet pentru generarea testelor este prezentat pe GitHub.

4.2 Specificațiile Sistemului de Calcul

Hardware

- **CPU:** Apple M1 (8-core CPU: 4 high-performance cores + 4 high-efficiency cores)
- **RAM:** 8 GiB Unified Memory
- **Storage:** 256 GiB High-Speed Custom SSD

Software

- **Operating System:** macOS Sequoia 15.1 (24B83)
- **Toolchain:** Python 3.14
- **Condiții de testare:** Toate nucleele CPU-ului operate la putere maximă, cu alocare maximă de memorie RAM.

4.3 Rezultatele evaluărilor

Pentru fiecare categorie de teste, am măsurat următoarele:

- **Timpul de execuție:** Durata necesară pentru a rezolva problema pentru fiecare instanță.
- **Calitatea soluției:** Valoarea totală obținută de algoritm comparată cu valoarea optimă (dacă este cunoscută).

Rezultatele au fost agregate și analizate separat pentru cele două categorii. Exemple de rezultate obținute:

Table 1. Rezultate pentru cazul 1, cu $G = 200$

#	n	**DP (ms)**	**Backtracking (ms)**	**Fractional (ms)**	**Accuracy (%)**
1	2	0.07	0.00	0.00	100.00
2	5	0.14	0.01	0.00	100.00
3	5	0.14	0.01	0.00	100.00
4	6	0.17	0.01	0.00	100.00
5	7	0.19	0.02	0.00	100.00
6	8	0.22	0.05	0.00	100.00
7	8	0.23	0.05	0.01	100.00
8	14	0.37	2.38	0.01	99.42
9	15	0.40	4.23	0.01	98.01
10	15	0.41	5.25	0.00	97.60
11	17	0.48	20.53	0.01	98.78
12	19	0.50	47.00	0.01	98.01
13	19	0.54	71.30	0.01	98.46
14	21	0.58	252.23	0.01	98.92
15	23	0.62	685.07	0.02	97.33
16	23	0.64	736.27	0.02	98.36
17	24	0.61	780.96	0.02	98.84
18	24	0.64	830.75	0.03	98.43
19	24	0.66	915.23	0.02	99.75
20	25	0.68	1,617.29	0.02	99.32
21	25	0.69	2,270.19	0.03	99.03
22	26	0.71	828.12	0.03	98.62
23	27	0.73	3,139.86	0.03	99.11
24	28	0.70	6,275.19	0.03	98.42
25	28	0.78	6,615.15	0.04	100.00
26	29	0.81	23,684.39	0.04	99.74
27	30	0.77	2,880.87	0.03	99.32
28	31	0.81	22,006.17	0.05	100.00
29	31	0.84	16,754.34	0.10	99.06

Table 2. Rezultate pentru cazul 2, cu $n = 32$

#	G	**DP (ms)**	**Backtracking (ms)**	**Fractional (ms)**	**Accuracy (%)**
1	6	0.02	0.02	0.01	94.01
2	14	0.04	0.03	0.01	89.32
3	35	0.11	0.68	0.01	97.42
4	39	0.12	1.77	0.01	97.72
5	39	0.13	1.95	0.01	94.18
6	60	0.21	5.18	0.01	97.05
7	60	0.22	5.61	0.01	97.74
8	66	0.29	13.73	0.01	95.90
9	70	0.26	6.26	0.01	99.19
10	71	0.30	13.31	0.02	97.35
11	77	0.29	47.40	0.02	99.46
12	87	0.34	150.16	0.02	97.50
13	94	0.37	107.14	0.02	98.73
14	94	0.37	149.58	0.02	98.63
15	102	0.40	318.34	0.07	98.55
16	105	0.45	202.40	0.02	98.63
17	116	0.50	409.47	0.03	99.48
18	129	0.54	1,394.17	0.03	99.26
19	138	0.56	2,220.77	0.03	97.51
20	138	0.61	2,237.25	0.03	99.53
21	146	0.61	2,733.64	0.03	99.02
22	156	0.67	1,278.30	0.03	98.65
23	156	0.67	8,647.90	0.03	98.35
24	163	0.67	12,004.60	0.02	98.76
25	170	0.72	5,117.95	0.02	98.55
26	171	0.71	11,301.76	0.03	98.93
27	175	0.82	33,803.91	0.04	99.73
28	186	0.77	4,114.02	0.03	98.21
29	189	0.79	12,319.65	0.11	99.28
30	191	0.82	22,240.62	0.03	98.39

4.4 Interpretarea valorilor obținute

În această secțiune analizăm performanța celor trei algoritmi testați (*Dynamic Programming - DP*, *Backtracking* și *Fractional Time*) pentru categoriile de probleme considerate.

Categoria 1

Algoritmul DP (Dynamic Programming). DP prezintă o creștere liniară a timpilor de execuție odată cu dimensiunea n, ceea ce indică o scalabilitate bună. Acesta este semnificativ mai rapid decât Backtracking, oferind soluții exacte într-un timp rezonabil. De exemplu, pentru $n = 8$, timpul de execuție este 0.22 ms, iar pentru $n = 25$, timpul scade la 0.69 ms.

Algoritmul Backtracking. Backtracking are timpi de execuție exponențiali, ceea ce îl face impracticabil pentru valori mari de n. De exemplu, pentru $n = 8$, timpul de execuție este 0.05 ms, comparativ cu 780 ms pentru $n = 24$. Aceasta confirmă complexitatea ridicată a acestui algoritm.

Algoritmul Fractional Time. Fractional Time este cel mai rapid dintre cei trei algoritmi, cu timpi foarte mici indiferent de dimensiunea n. Spre exemplu, pentru $n = 19$, timpul de execuție este 0.01 ms, iar pentru $n = 31$, timpul este de 0.1 ms. Acest lucru sugerează că algoritmul folosește o metodă euristică, aproximativă, ideală pentru aplicații unde viteza este critică.

Categoria 2

Algoritmul DP (Dynamic Programming). În această categorie, DP continuă să fie rapid și scalabil. Timpul de execuție crește liniar cu valoarea G. De exemplu, pentru $G = 6$, timpul este 0.02 nm, iar pentru $G = 102$, acesta crește la 0.4 ms.

Algoritmul Backtracking. Timpul de execuție al algoritmului Backtracking este puternic influențat de creșterea valorii G, ceea ce îl face dificil de utilizat pentru valori mari. De exemplu, pentru $G = 6$, timpul este 0.02 ms, iar pentru $G = 102$, acesta ajunge la 318 ms.

Algoritmul Fractional Time. Fractional Time rămâne algoritmul cel mai rapid, chiar și în această categorie. Creșterea valorii G are un impact moderat asupra timpului de execuție. De exemplu, pentru $G = 6$, timpul este 0.01 ms, iar pentru $G = 102$, timpul crește ușor la 0.07 ms.

5 Concluzii Generale

- **Fractional Time** este cel mai rapid algoritm și este potrivit pentru aplicații unde viteza este prioritară, chiar dacă soluțiile sunt aproximative.
- **Dynamic Programming** oferă un compromis excelent între viteză și acuratețe, fiind scalabil și eficient pentru probleme mai mari.
- **Backtracking** este cel mai lent algoritm, fiind potrivit doar pentru probleme mici sau pentru explorări exhaustive unde sunt necesare soluții exacte.

În concluzie, algoritmul demonstrează o performanță robustă pentru instanțele generate, oferind o bază solidă pentru testarea unor optimizări ulterioare.

6 Bibliografie

- https://www.pbinfo.ro/probleme/1886/rucsac1
- https://www.infoarena.ro/problema/rucsac
- https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/