Knapsack Problem

Butucianu Marius, Iesanu Sorin, Ursescu Sebastian - 325CA

University Politehnica of Bucharest

1 Introducere

1.1 Descrierea problemei

Problema rucsacului (Knapsack problem) este un exemplu clasic în teoria optimizării combinatorii și se referă la selectarea unui subset de obiecte, cu scopul de a maximiza valoarea totală, sub restricția unui anumit volum sau greutate. Problema poate fi formulată astfel: avem un rucsac cu o capacitate limitată și o serie de obiecte, fiecare cu o valoare și o greutate asociate. Scopul este de a alege obiectele care vor fi puse în rucsac, astfel încât valoarea totală să fie maximă, iar greutatea totală să nu depășească capacitatea rucsacului.

În ziua de astăzi, problema rucsacului se regăsește, în principal, în domeniul financiar, dar și în alte domenii precum gestionarea resurselor etc. Pentru un investitor este de cel mai mare interes optimizarea modului în care îți distribuie capitalul de-a lungul numeroaselor sale investiții pentru ca, în același timp, să minimizeze costul investițiilor inițiale și să își maximizeze profitul obținut. De asemenea, multe companii caută să gestioneze optim diferitele canale de marketing (Google Ads, reclame TV, sponzorizarea unor celebrități, reclame pe diverse platforme de social media), astfel încât să poată stârni interesul a cât mai multor persoane. Este o problemă de care se lovește orice persoană, cel puțin săptămânal: care este cel mai bun mod de folosire a salariului greu câștigat pentru a supraviețui, pentru a putea economisii și chiar pentru a rămâne puțin și pentru vacanțe.

2 Demonstrație NP-Hard

Reducerea de la 3-SAT la Problema Rucsacului
Pentru a demonstra că Problema Rucsacului este NP-Hard, vom efectua o reducere în timp polinomial de la problema 3-SAT (o problemă NP-Completă bine cunoscută) la Problema Rucsacului.

2.1. Problema 3-SAT
Problema 3-SAT constă în a decide dacă o formulă Booleană FF, exprimată în formă normală conjunctivă (CNF), este satisfiabilă. Formula este de forma:
F=$(C_1 \wedge C_2 \wedge \dots C_m)$
unde fiecare clauză Cj conține exact 3 literali (de exemplu, $x_1 \vee !x_2 \vee x_3$)

Obiectivul este să găsim o atribuire de valori adevărat/fals variabilelor astfel încât formula să fie satisfăcută.

2.2. Codificarea problemei 3-SAT în Problema Rucsacului

Vom transforma formula 3-SAT într-o instanță a Problemei Rucsacului, astfel încât soluționarea Problemei Rucsacului să ofere o soluție pentru formula 3-SAT.
Pașii codificării:

1. Literal și variabilă: Fiecărui literal din formula 3-SAT îi corespunde un obiect în Problema Rucsacului.
    - Fiecare obiect are o greutate w_i și o valoare v_i
2. Clauze și satisfiabilitate: Fiecare clauză C_i trebuie să fie satisfăcută. Vom construi greutăți și valori astfel încât alegerea unui subset de obiecte să satisfacă cel puțin o variabilă din fiecare clauză.
3. Capacitatea rucsacului W: Capacitatea rucsacului este setată pentru a permite alegerea unui subset valid care satisface toate clauzele.
4. Valoarea țintă V: Valoarea totală a obiectelor selectate trebuie să atingă o valoare țintă V, corespunzătoare unei soluții satisfiabile pentru formula 3-SAT.

2.3. Construirea instanței pentru Problema Rucsacului

Pentru fiecare clauză C_j :

- Construim un set de obiecte care reprezintă literalii din clauză.
- Alegerea unui obiect (literal) din fiecare clauză reprezintă satisfacerea clauzei respective.

Parametrii:

- Greutățile w_i și valorile v_i sunt alese astfel încât:
    - Greutățile obiectelor să fie distincte.
    - Alegerea unui literal (obiect) să contribuie la satisfacerea clauzei fără a depăși capacitatea rucsacului.

4. Verificarea reducerii

- Dacă formula 3-SAT este satisfiabilă, atunci putem selecta un subset de obiecte care respectă constrângerile de greutate ale rucsacului și care ating valoarea țintă V.
- Dacă Problema Rucsacului are o soluție validă (adică un subset care atinge valoarea V fără a depăși capacitatea W), atunci există o atribuire a variabilelor care satisface formula 3-SAT.

2.5. Complexitatea reducerii

Codificarea formulei 3-SAT într-o instanță a Problemei Rucsacului se face în timp polinomial, deoarece transformarea greutăților, valorilor și a capacității rucsacului implică doar calcule simple și construcții directe.

Concluzie

Deoarece am realizat o reducere în timp polinomial de la 3-SAT (o problemă NP-Completă) la Problema Rucsacului, aceasta din urmă este cel puțin la fel de dificilă ca 3-SAT. Prin urmare, Problema Rucsacului este NP-Hard.

3 Algoritmii utilizați în rezolvarea problemei

3.1 Backtracking optimizat

Algoritmul implementat folosește o combinație de backtracking și memoizare pentru a rezolva problema Knapsack. Obiectivul este de a determina cea mai mare valoare totală ce poate fi obținută dintr-un set de obiecte, respectând constrângerea greutății maxime. Strategia implică explorarea tuturor combinațiilor posibile de obiecte, evitând recalculările inutile prin memoizare.

Memoizarea: O matrice 2D de tip MemoEntry este folosită pentru a stoca valorile calculate anterior, reducând astfel timpul de execuție pentru subprobleme recurente. Fiecare intrare din matrice conține un flag valid și valoarea asociată.
Funcția de backtracking începe explorarea de la primul obiect (în ordinea sortării) și evaluează două scenarii pentru fiecare obiect:

- Include obiectul curent: Doar dacă greutatea totală rămâne sub limita permisă.
- Exclude obiectul curent: Trecerea la următorul obiect fără a-l in-clude pe cel curent.

Valoarea maximă este determinată alegând soluția optimă dintre cele două scenarii.
Sortare: Obiectele sunt sortate în funcție de raportul valoare-greutate pentru a prioritiza includerea obiectelor mai valoroase și mai ușoare.
Actualizarea soluției optime: Când se găsește o combinație cu valoare mai mare decât soluția curentă, aceasta este salvată.

Complexitatea teoretică

- **Timp**:
  - Fără memoizare: Complexitatea este O(2^n) în cel mai rău caz, unde n este numărul de obiecte.
  - Cu memoizare: Complexitatea scade la O(n * W), unde W este greutatea maximă permisă.
- **Spațiu**: O(n * W) pentru stocarea tabelului de memoizare.

3.2 Sortare greedy

Acest algoritm aproximează soluția prin sortarea vectorului de obiecte, folosind o formulă ce ia în considerare atât raportul dintre valoarea obiectului și greutatea sa, cât și raportul dintre greutatea obiectului și capacitatea maximă a rucsacului, astfel, obiectele ce au un raport mare valoare / greutate vor fi prioritizate, iar cele ce ocupă un spațiu prea mare în comparație cu rucsacul vor tinde înspre a fi excluse.

Algoritmul oferă rezultate bune pentru obiecte cu greutate și valoare mică, iar pentru obiecte cu greutate și valoare mai mare, rezultatele lasă de dorit. Principalul avantaj este timpul relativ scurt de rulare (în comparație cu rezolvarea „brute-force", ce constă într-o abordare de backtracking clasică, ce rulează în mereu în timp exponențial, algoritmul va rula, în cel mai rău caz, în timp polinomial, iar în medie, rulează în timp quasiliniară).

Pentru analizarea complexității algoritmului, se pune în discuție analizarea celor 2 funcții din care este compus: **objectwvcomp** și **high_value**. Întrucât funcția objectwvcomp este formată doar din structuri de decizie și operații aritmetice **simple**, această funcție are complexitate constantă în timp și spațiu. Pentru analiza funcției high_value, este necesară cunoașterea implementării funcției **qsort**[1]. Din fericire, biblioteca libc care oferă implementarea, garantează faptul că această funcție va rula în medie în timp quasiliniar ($\mathbf{O}(nlog(n))$), iar în cel mai rău caz timpul de rulare este polinomial ($\mathbf{O}(n^2)$). De asemnea, libc garantează că spațiul ocupat de această funcție este logaritmic ($\mathbf{O}(log(n))$). În final, algoritmul va trece în cel mai rău caz prin toate obiectele disponibile, iar în medie va procesa cel puțin atâtea obiecte câte sunt în soluție, astfel adăugând complexitate spațială constantă ($\mathbf{O(1)}$) și complexitate temporală liniară ($\mathbf{O}(n)$). În final, complexitatea algoritmului este: **logaritmică** din punct de vedere **spațial** și **quasiliniară** din punct de vedere temporal.

4 Evaluarea soluției

4.1 Setul de date

Pentru verificarea corectitudinii rezolvării, au fost folosite 157 de teste generate aleator, împărțite în 4 categorii distincte: **număr mare** de obiecte cu greutate și valoare **mare**, **număr mare** de obiecte cu greutate și valoare **mică**, **număr mic** de obiecte cu greutate și valoare **mare** și teste cu un **număr mic** de obiecte cu greutate și valoare **mică**. De asemenea, pentru o mică garanție a veridicității rezultatelor obținute, câteva din testele mici au fost verificate folosind un algoritm de „brute force". Algoritmul cu rezultatele apropiate cel mai mult de valorile reale este cel ce folosește backtracking, diferențele fiind aproape neglijabile.

4.2 Specificațiile sistemului

Rularea testelor a fost realizată pe un sistem Linux Manjaro dotat cu: procesor AMD 5800x3D și 16 GB RAM cu o viteză de 4800 MT/s. Algoritmii au fost implementați în limbajul C și au fost compilați cu GCC, folosind flag-ul de optimizare maximă (O3), pe același sistem de operare.

4.3 Rezultate

Teste cu un număr mic de obiecte cu greutate și valoare mare:

Backtracking:

Nr. Obiecte	**Valoare**	**Timp(s)**
6	2729	0.001
8	3819	0.001
27	9096	0.002
26	10397	0.002
13	6599	0.001
15	8473	0.001
17	7059	0.001
14	6853	0.001
26	9305	0.001
14	6958	0.001

Greedy:

Numărul de obiecte	Valoarea optimă găsită	Timpul de rulare
6	2893	0.002
8	4054	0.001
27	8470	0.001
26	9968	0.001
13	6599	0.001
15	8473	0.001
17	7059	0.001
14	6704	0.001
26	9269	0.001
14	6216	0.001

Teste cu un număr mic de obiecte cu greutate și valoare mică:

Backtracking:

Nr. Obiecte	**Valoare**	**Timp(s)**
2	23	0.001
8	141	0.001
20	1716	0.001
2	62	0.001
3	52	0.001

8	426	0.001
6	855	0.001
17	2527	0.001
8	358	0.001
11	1364	0.001
16	1826	0.001
6	141	0.001
8	269	0.001
19	2093	0.001
4	210	0.001
13	2133	0.001
4	176	0.001
9	264	0.001
3	134	0.001
5	90	0.001
3	144	0.001
19	2863	0.001
6	264	0.001
14	1664	0.001
13	968	0.001
18	1677	0.001
19	2474	0.001
2	61	0.001
15	1708	0.001
9	394	0.001
7	311	0.001
2	16	0.001
8	279	0.001
10	152	0.001
10	338	0.001
4	98	0.001
7	142	0.001
5	160	0.001
10	252	0.001
8	272	0.001
13	2331	0.001
16	3089	0.001
6	301	0.001

9	1354	0.001
6	910	0.001
7	144	0.002
13	1419	0.002
4	139	0.002
2	50	0.002
12	1452	0.001
6	151	0.001
10	105	0.001
16	1851	0.001
5	1188	0.001
8	282	0.001
2	87	0.001
2	72	0.001

Greedy:

Numărul de obiecte	Valoarea optimă găsită	Timpul de rulare
2	30	0.001
8	141	0.002
20	1383	0.001
2	62	0.002
3	57	0.002
8	426	0.001
6	855	0.001
17	2403	0.002
8	358	0.001
11	1133	0.001
16	1505	0.001
6	141	0.001
8	269	0.001
19	2050	0.002
4	272	0.002
13	1804	0.002
4	184	0.001
9	264	0.001
3	134	0.001
5	90	0.001

3	154	0.001
19	2679	0.001
6	264	0.001
14	1664	0.001
13	878	0.001
18	1273	0.001
19	1850	0.001
2	129	0.001
15	1547	0.001
9	394	0.001
7	311	0.001
2	32	0.001
8	261	0.001
10	152	0.001
10	338	0.001
4	98	0.001
7	90	0.001
5	92	0.001
10	247	0.001
8	275	0.001
13	2331	0.001
16	3089	0.001
6	301	0.001
9	1166	0.001
6	910	0.001
7	144	0.001
13	1419	0.001
4	148	0.001
2	74	0.001
12	1309	0.001
6	151	0.001
10	105	0.001
16	1602	0.001
5	1323	0.001
8	282	0.001
2	87	0.001
2	91	0.001

Teste cu un număr mare de obiecte cu greutate și valoare mică:

Backtracking:

Nr. Obiecte	**Valoare**	**Timp(s)**
9366	18478	4.899
7120	15594	3.356
5501	13262	1.450
8606	15764	0.845
7654	15635	3.754
9805	18146	2.352
8679	15574	0.776
6712	15140	2.047
6859	14575	2.245
6738	14384	1.909
8203	16509	2.889
8818	17638	2.955
6000	14990	1.352
8398	16468	2.940
9672	19645	1.805
7750	16213	2.651
6899	16739	3.427
7947	15330	1.640
6257	13209	1.311
7646	16437	0.723

Greedy:

Numărul de obiecte	Valoarea optimă găsită	Timpul de rulare
9366	17496	0.004
7120	14971	0.003
5501	12743	0.002
8606	14775	0.003
7654	14805	0.003
9805	17008	0.004
8679	14492	0.003
6712	14325	0.003
6859	13787	0.002

6738	13633	0.002
8203	15640	0.003
8818	16645	0.003
6000	14277	0.002
8398	15626	0.003
9672	18664	0.003
7750	15199	0.002
6899	15810	0.003
7947	14457	0.003
6257	12544	0.002
7646	15689	0.003

Teste cu un număr mare de obiecte cu greutate și valoare mare:

Backtracking:

Nr. Obiecte	**Valoare**	**Timp(s)**
1167	81909	0.512
1783	103871	1.302
1043	81877	0.463
1716	107192	1.284
1982	115750	1.544
1692	107337	0.554
1696	112835	1.291
1332	99165	0.942
1341	99161	0.535
1558	98102	1.143
1430	90420	0.906
1350	94559	0.948
1341	98657	0.442
1341	94318	0.471
1852	105417	0.842
1354	96110	0.574
1658	108427	0.775
1765	109155	1.023
1393	103948	0.929
1375	97553	0.585
1495	93986	0.657
1311	95210	0.979

1247	87173	0.765
1462	98991	0.945
1328	88744	0.581
1342	92115	0.429
1392	96163	0.500
1119	75172	0.264
1499	101025	0.628
1264	98025	0.559
1165	91008	0.289
1907	105382	1.406
1807	104874	0.846
1749	124876	0.861
1519	99950	0.570
1114	87640	0.583
1368	101141	0.836
1668	105820	0.532
1763	109041	0.520
1356	96689	0.807
1492	95170	0.960
1413	98806	0.450
1973	105191	0.661
1531	96796	0.667
1657	107784	0.750
1474	105293	0.682
1094	88202	0.685
1071	92590	0.609
1395	91829	0.546
1421	105884	0.563
1245	88172	0.736
1079	80022	0.645
1137	81402	0.682
1220	82519	0.652
1613	106135	0.638
1194	91257	0.344
1997	111913	1.309
1532	102028	0.476
1286	92747	0.492
1578	102644	0.479

Greedy:

Numărul de obiecte	Valoarea optimă găsită	Timpul de rulare
1167	44563	0.001
1783	61791	0.001
1043	48941	0.001
1716	66606	0.001
1982	79557	0.001
1692	63752	0.001
1696	78263	0.001
1332	58276	0.001
1341	61615	0.001
1558	62587	0.001
1430	52100	0.001
1350	59878	0.001
1341	68104	0.001
1341	58673	0.001
1852	70105	0.002
1354	60033	0.001
1658	68359	0.002
1765	73236	0.002
1393	67226	0.002
1375	62879	0.002
1495	51901	0.002
1311	64955	0.002
1247	57857	0.002
1462	59049	0.002
1328	55350	0.002
1342	49661	0.002
1392	55100	0.002
1119	44481	0.002
1499	61525	0.003
1264	63984	0.002
1165	51057	0.002
1907	65484	0.003
1807	63791	0.002
1749	83253	0.002

1519	61263	0.002
1114	56705	0.001
1368	59575	0.001
1668	60944	0.001
1763	75682	0.001
1356	58860	0.001
1492	60419	0.001
1413	59014	0.001
1973	65451	0.001
1531	60508	0.001
1657	67619	0.001
1474	53934	0.001
1094	53626	0.001
1071	59169	0.001
1395	59028	0.001
1421	61116	0.001
1245	53903	0.001
1079	44728	0.001
1137	48078	0.001
1220	51876	0.001
1613	67639	0.001
1194	63207	0.001
1997	76716	0.001
1532	62120	0.001
1286	60980	0.001
1578	69087	0.001

Teste cu un număr mai mare de obiecte cu greutate și valoare mare:

Backtracking:

Nr. Obiecte	**Valoare**	**Timp(s)**
8978	247497	109.842
5228	186798	63.433
1162	86741	9.975
9754	245099	138.604
2874	131185	30.388
5135	189223	49.093
6255	202139	94.748

8150	234105	80.264
3411	158588	41.551
4816	185532	66.315

Greedy:

Numărul de obiecte	Valoarea optimă găsită	Timpul de rulare
8978	211575	0.004
5228	143438	0.003
1162	52324	0.001
9754	207263	0.003
2874	84734	0.001
5135	151017	0.002
6255	159051	0.002
8150	197166	0.002
3411	120603	0.002
4816	145025	0.002

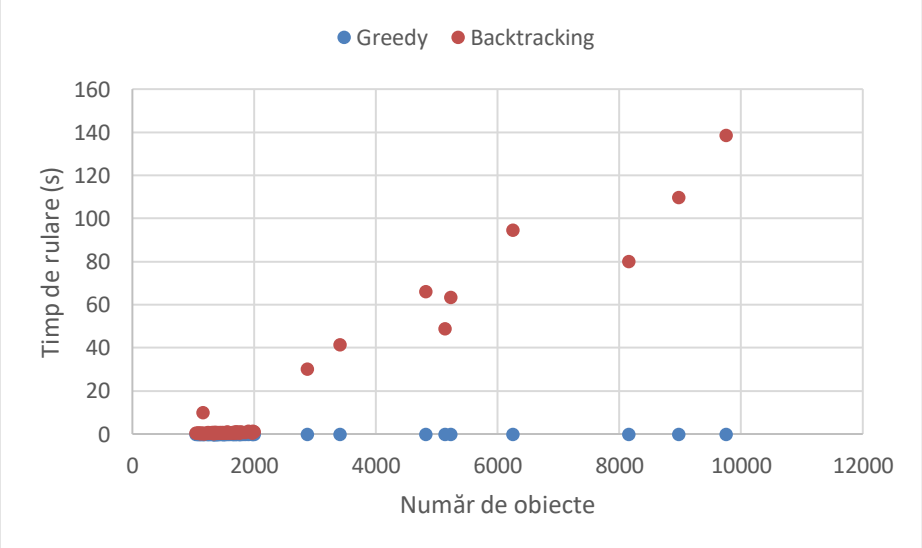

Fig. 1. Reprezentare grafică a timpului de rulare in funcție de numărulde obiecte din test (pentru testele cu un număr mare de obiecte)

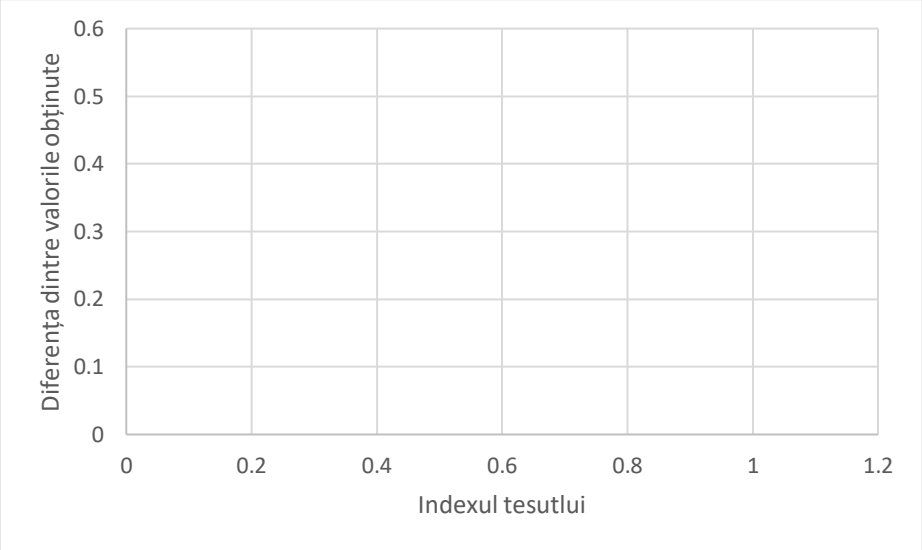

Fig. 2. Reprezentare grafică a diferenței dintre valorile obținute (pentru testele cu un număr mare de obiecte)

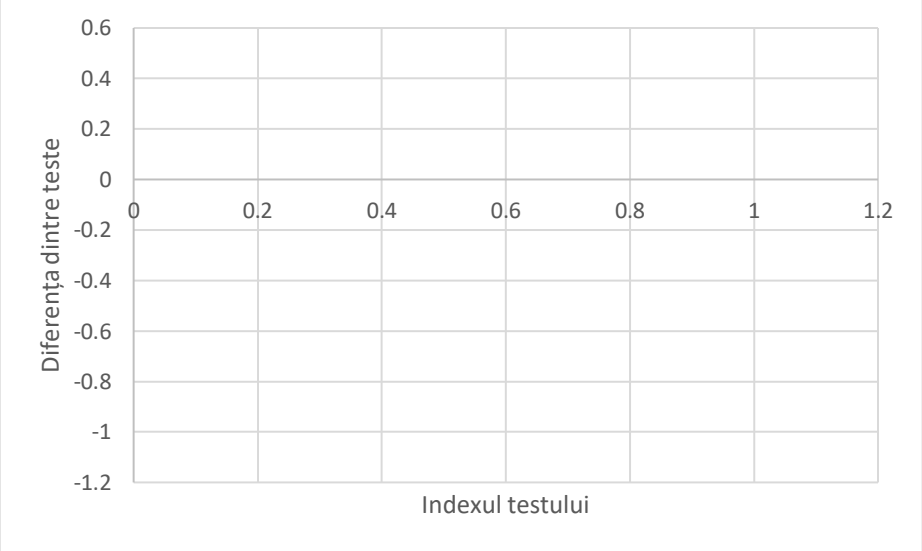

Fig. 3. Reprezentare grafică a diferenței dintre valorile obținute, sub formă de procent (pentru testele cu un număr mic de obiecte)

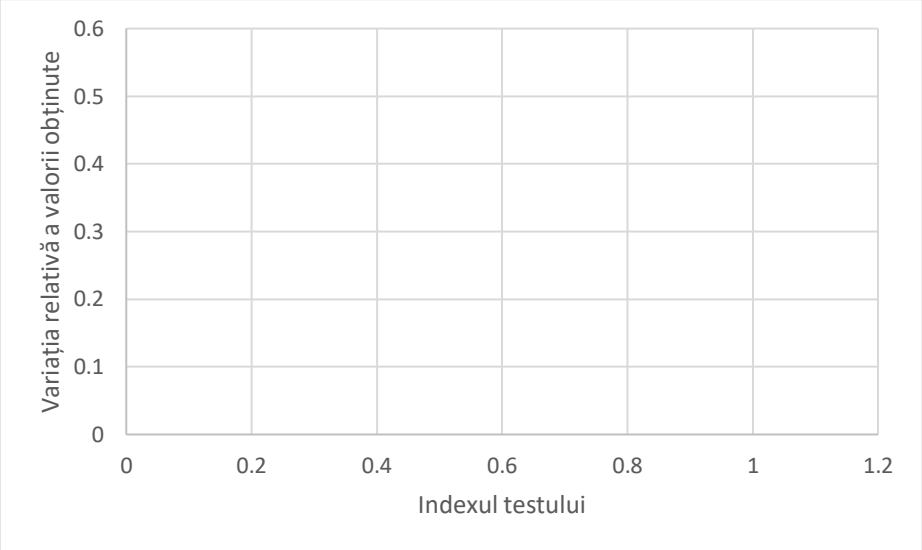

Fig. 4. Reprezentare grafică a diferenței relative dintre valorile obținute, sub formă de procent (pentru testele cu un număr mare de obiecte)

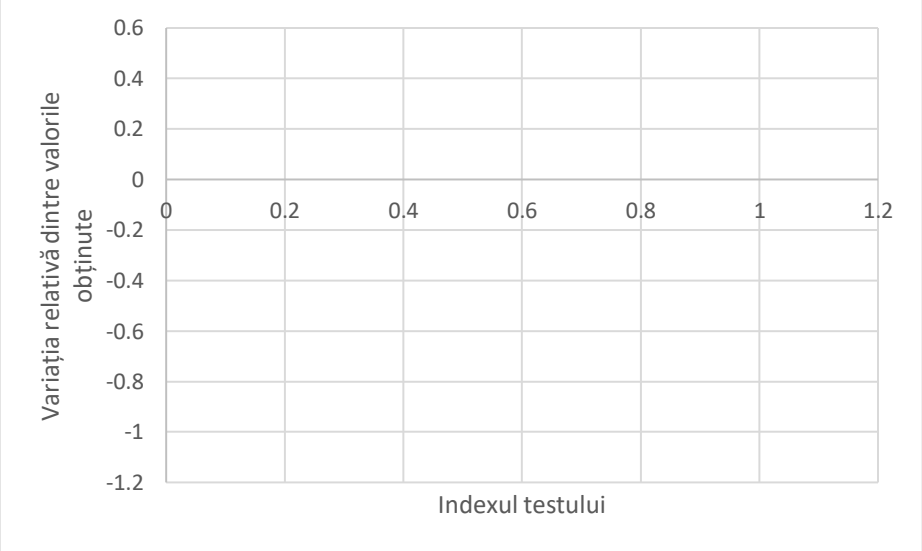

Fig. 5. Reprezentare grafică a diferenței relative dintre valorile obținute, sub formă de procent (pentru testele cu un număr mic de obiecte)

5 Concluzii

Se poate afirma că orice algoritm care calculează valoarea optimă a rucsacului, într-un timp rezonabil, nu poate decât să aproximeze această valoare, întrucât singura rezolvare care oferă de fiecare dată un răspuns exact este abordarea clasică prin backtracking neoptimizat, însă marile dezavantaje al acestei abordări sunt: complexitatea temporală exponențială și complexitatea spațială polinomială, astfel, pentru seturi de date suficient de mari, această abordare este prea ineficientă pentru a putea fi folosită.

În prisma rezultatelor obținute, pentru rezolvarea problemei rucsacului avem la dispoziție două abordări principale: o variantă de backtracking optimizată, ce renunță la anumite permutări pentru a putea finaliza calculul mai rapid, dar fără a sacrifica prea mult acuratețea soluției și o variantă de sortare și formare a soluției, sacrificând mai mult acuratețea soluției pentru a o putea obține foarte rapid.

References

1. https://elixir.bootlin.com/glibc/glibc-2.40.9000/source/stdlib/qsort.c