Subset Sum Problem: NP-Hardness, Algoritmi și Evaluare Experimentală

Ioan Ursescu, Ionuț Mihăilescu, and Mihuț Matei

Universitatea Națională de Știință și Tehnologie Politehnica București
Facultatea de Automatică și Calculatoare

Abstract. Problema *Subset Sum* reprezintă una dintre problemele fundamentale din teoria complexității computaționale, fiind un caz particular al problemei *Knapsack* și un exemplu clasic de problemă *NP-Complete*. În această lucrare este analizată problema Subset Sum atât din perspectivă teoretică, cât și experimentală. Sunt prezentate definiția formală a problemei și demonstrația caracterului său NP-Complete, printr-o reducere polinomială de la o problemă NP-Hard cunoscută.
În continuare, sunt descriși și analizați mai mulți algoritmi utilizați pentru rezolvarea problemei Subset Sum, incluzând metode exponențiale de tip backtracking, algoritmi pseudo-polinomiali bazați pe programare dinamică și tehnici de tip *meta-euristici stocastice*. Pentru fiecare abordare sunt discutate complexitatea computațională, avantajele și limitările practice.
Partea experimentală a lucrării constă în evaluarea performanței algoritmilor propuși pe un set de teste variat, construit pentru a evidenția comportamentul acestora în funcție de dimensiunea instanțelor și de valorile numerice implicate. Rezultatele obținute sunt analizate comparativ și permit formularea unor recomandări privind alegerea metodei adecvate în funcție de contextul practic. Lucrarea evidențiază astfel diferențele dintre fezabilitatea teoretică și eficiența practică a soluțiilor pentru problema Subset Sum.

Keywords: Subset Sum · NP-Complete · Dynamic Programming · Genetic Algorithms · Approximation

1 Introducere

1.1 Descrierea problemei rezolvate

Problema *Subset Sum* reprezintă una dintre cele mai fundamentale probleme din teoria complexității computaționale și optimizării combinatoriale, fiind un caz particular al problemei *Knapsack*. În forma sa clasică, problema este definită astfel: fie o mulțime finită de numere întregi nenegative $A = \{a_1, a_2, \ldots, a_n\}$ și o valoare țintă $T \in \mathbb{N}$. Se cere să se determine dacă există o submulțime $S \subseteq A$ astfel încât suma elementelor sale să fie egală cu valoarea țintă, adică

$$\sum_{a \in S} a = T.$$

Varianta de *decizie* a problemei Subset Sum este cunoscută ca fiind *NP-Complete*, ceea ce implică faptul că, deși o soluție propusă poate fi verificată în timp polinomial, nu este cunoscut niciun algoritm care să rezolve toate instanțele problemei în timp polinomial, presupunând că $P \neq NP$. Cu toate acestea, anumite instanțe restrânse pot fi abordate eficient prin algoritmi pseudo-polinomiali, în special atunci când valoarea țintă T este relativ mică în raport cu dimensiunea intrării.

Studiul problemei Subset Sum este relevant atât din perspectivă teoretică, prin rolul său central în teoria complexității, cât și din perspectivă practică, prin multitudinea de contexte în care aceasta poate fi utilizată ca model abstract pentru probleme reale de decizie și optimizare.

1.2 Aplicații practice

Problema *Subset Sum* apare în mod natural într-o varietate de contexte practice în care se impune selectarea unui subansamblu de elemente astfel încât suma unor valori asociate acestora să respecte o constrângere exactă. Un domeniu important de aplicare este alocarea resurselor, unde se dorește alegerea unui set de activități, procese sau componente astfel încât consumul total de resurse (timp, energie sau cost) să fie egal cu un buget prestabilit, situații frecvent întâlnite în planificarea proiectelor și managementul operațional.

În planificarea financiară și gestionarea bugetelor, problema Subset Sum poate modela scenarii în care este necesară selectarea unui subset de cheltuieli sau investiții care să însumeze exact o valoare impusă. Exemple concrete includ distribuirea fondurilor între mai multe categorii de cost sau verificarea posibilității de echilibrare a unor conturi prin combinarea unor sume discrete disponibile.

În domeniul securității informației, problema Subset Sum a stat la baza unor scheme criptografice timpurii de tip *knapsack*, care exploatau dificultatea rezolvării problemei în cazul general. Deși aceste sisteme nu mai sunt utilizate pe scară largă din cauza vulnerabilităților descoperite ulterior, ele evidențiază relevanța problemei în proiectarea și analiza mecanismelor criptografice.

Problema apare, de asemenea, în selecția și configurarea sistemelor, unde este necesară alegerea unui set de componente compatibile astfel încât suma caracteristicilor acestora (de exemplu, cost, consum energetic sau capacitate) să respecte o limită exactă. Astfel de aplicații pot fi întâlnite în configurarea sistemelor embedded, proiectarea infrastructurilor software sau dimensionarea resurselor hardware.

Nu în ultimul rând, Subset Sum este frecvent utilizată ca problemă intermediară în modelarea și demonstrarea NP-dificultății pentru alte probleme combinatoriale, ceea ce subliniază, pe lângă relevanța sa practică, importanța sa teoretică.

2 Demonstrație NP-Complete

În această secțiune demonstrăm că varianta de decizie a problemei *Subset Sum* este **NP-Complete**. Demonstrația urmează structura clasică: (i) arătăm că problema aparține clasei NP și (ii) demonstrăm că este NP-hard printr-o reducere polinomială de la o problemă NP-Complete cunoscută.

2.1 Încadrarea în NP

Considerăm varianta de *decizie* a problemei *Subset Sum*: dată o mulțime finită de numere întregi nenegative $A = \{a_1, a_2, \ldots, a_n\}$ și o valoare țintă $T \in \mathbb{N}$, se decide dacă există o submulțime $S \subseteq A$ astfel încât

$$\sum_{a \in S} a = T.$$

Pentru a demonstra că problema *Subset Sum* aparține clasei NP, identificăm un certificat și o procedură de verificare în timp polinomial. Un certificat natural este un vector binar $x \in \{0,1\}^n$, unde $x_i = 1$ indică faptul că elementul a_i este inclus în submulțimea propusă.

Procedura de verificare calculează suma

$$\Sigma(x) = \sum_{i=1}^{n} x_i \cdot a_i$$

și acceptă certificatul dacă și numai dacă $\Sigma(x) = T$. Calculul sumei necesită n operații aritmetice elementare, deci timpul de verificare este $\mathcal{O}(n)$, polinomial în dimensiunea intrării. Prin urmare, problema *Subset Sum* aparține clasei NP.

2.2 Reducere de la o problemă NP-Hard cunoscută

Pentru a demonstra NP-hardness, construim o reducere polinomială de la problema *PARTITION*, cunoscută ca fiind NP-Complete.

Definiția problemei PARTITION. Dată o mulțime (sau multiset) de numere întregi nenegative $A = \{a_1, a_2, \ldots, a_n\}$, se decide dacă A poate fi împărțită în două submulțimi disjuncte A_1 și A_2 astfel încât

$$\sum_{a \in A_1} a = \sum_{a \in A_2} a.$$

Definiția problemei-țintă. Problema *Subset Sum* (decizie): dată o mulțime $A = \{a_1, a_2, \ldots, a_n\}$ și o valoare T, se decide dacă există $S \subseteq A$ cu

$$\sum_{a \in S} a = T.$$

Transformarea. Fie o instanță a problemei PARTITION definită pe mulțimea A. Calculăm suma totală a elementelor:

$$W = \sum_{i=1}^{n} a_i.$$

Dacă W este impar, instanța PARTITION este în mod evident falsă. În caz contrar, construim instanța Subset Sum folosind aceeași mulțime A și alegem valoarea țintă

$$T = \frac{W}{2}.$$

Transformarea presupune doar calculul sumei W și al valorii T, fiind realizabilă în timp polinomial.

Corectitudinea reducerii. ($\Rightarrow$) Presupunem că instanța PARTITION este adevărată. Atunci există o partiție $A = A_1 \cup A_2$, cu $A_1 \cap A_2 = \emptyset$, astfel încât $\sum_{a \in A_1} a = \sum_{a \in A_2} a = W/2$. Rezultă că submulțimea $S = A_1$ satisface $\sum_{a \in S} a = T$, deci instanța Subset Sum este adevărată.

($\Leftarrow$) Presupunem că instanța Subset Sum este adevărată. Atunci există $S \subseteq A$ cu $\sum_{a \in S} a = T = W/2$. Considerăm complementul $A \setminus S$. Atunci

$$\sum_{a \in A \setminus S} a = W - \sum_{a \in S} a = W - \frac{W}{2} = \frac{W}{2},$$

deci S și $A \setminus S$ definesc o partiție validă a lui A. Prin urmare, instanța PARTITION este adevărată.

Concluzie. Am demonstrat că problema *Subset Sum* aparține clasei NP și că este NP-hard printr-o reducere polinomială de la problema *PARTITION*. Întrucât sunt îndeplinite ambele condiții, rezultă că varianta de decizie a problemei *Subset Sum* este **NP-Complete**.

3 Prezentare Algoritmi

Pentru evitarea ambiguității, vom considera N - numărul de elemente din mulțime și T - suma țintă pentru care trebuie să găsim o submulțime.

3.1 Descrierea algoritmilor propuși

3.1.1 Backtracking

Abordarea de tip **backtracking** (căutare exhaustivă) rezolvă varianta de decizie a problemei *Subset Sum* explorând spațiul tuturor submulțimilor posibile. Algoritmul parcurge elementele mulțimii în ordine și, pentru fiecare element a_i, ia una dintre cele două decizii:

- elementul este **exclus** din submulțime;
- elementul este **inclus** în submulțime, iar suma rămasă scade cu a_i.

Formal, definim o funcție recursivă $\mathrm{BT}(i, rem)$ care întoarce adevărat dacă există o submulțime a elementelor din intervalul $\{a_i, \ldots, a_{n-1}\}$ cu sumă egală cu *rem*. Cazurile de bază sunt:

- dacă $rem = 0$, atunci am găsit o soluție (submulțimea curentă este martor);
- dacă $i = n$, nu mai există elemente disponibile, deci nu mai putem construi soluția.

În implementare se menține un vector `subset` care reține elementele selectate până în acel punct (martorul soluției). La includerea lui a_i, elementul este adăugat în `subset`, iar după revenirea din recursie este eliminat (operație *push/pop*) pentru a restaura starea inițială.

Metoda este **exactă** și returnează prima soluție găsită (dacă există), însă în lipsa unor tehnici de tăiere (*pruning*) poate explora un număr exponențial de ramuri.

3.1.2 Dynamic Programming

Algoritmul de programare dinamică implementat pentru rezolvarea problemei *Subset Sum* utilizează o abordare de tip **top-down** (recursivă) cu **memoizare**, extinsă pentru a permite tratarea valorilor negative în mulțimea de intrare. Scopul algoritmului este determinarea existenței unui subansamblu ale cărui elemente însumează exact valoarea țintă T.

Ideea fundamentală constă în luarea, pentru fiecare element, a unei decizii binare: includerea sau excluderea acestuia din submulțime. Problema este formalizată printr-o funcție de decizie definită ca:

$$DP(n, S) = \begin{cases} \text{adevărat}, & \text{dacă există un subset al primelor } n \text{ elemente cu suma } S, \\ \text{fals}, & \text{altfel.} \end{cases}$$

Tratarea valorilor negative. Spre deosebire de formularea clasică, care presupune valori nenegative, această implementare permite valori negative. Pentru a evita utilizarea unor indici negativi în tabelul de programare dinamică, se determină suma minimă posibilă $minSum$ (suma tuturor valorilor negative) și suma maximă posibilă $maxSum$ (suma tuturor valorilor pozitive). Domeniul tuturor sumelor realizabile este astfel:

$$[minSum,\ maxSum].$$

Se introduce un **offset** definit ca $offset = -minSum$, astfel încât orice sumă S este mapată la un indice valid prin $S + offset$.

Structura tabelului DP. Tabelul de memoizare este definit ca un tablou bidimensional

$$memo[n][S + offset],$$

unde n reprezintă numărul de elemente considerate, iar valoarea din tabel este 1 dacă suma este realizabilă, 0 dacă nu este realizabilă și -1 dacă starea nu a fost încă evaluată.

Cazuri de bază. Dacă suma curentă este 0, există întotdeauna o soluție reprezentată de submulțimea vidă. În schimb, dacă nu mai există elemente disponibile ($n = 0$), singura sumă realizabilă este 0, toate celelalte fiind imposibile.

Relația de recurență. Pentru $n > 0$, o sumă S este realizabilă dacă cel puțin una dintre următoarele două situații este adevărată:

- elementul a_{n-1} este exclus, iar suma S este realizabilă cu primele $n - 1$ elemente;
- elementul a_{n-1} este inclus, iar suma $S - a_{n-1}$ este realizabilă cu primele $n - 1$ elemente.

Această observație conduce la relația de recurență:

$$DP(n, S) = DP(n - 1, S) \vee DP(n - 1, S - a_{n-1}).$$

Rezultatele intermediare sunt memorate în tabelul DP pentru a evita recalculările, reducând astfel semnificativ complexitatea în timp comparativ cu o abordare recursivă naivă.

Reconstrucția soluției. După determinarea existenței unei soluții pentru suma T, algoritmul reconstruiește submulțimea corespunzătoare prin parcurgerea inversă a tabelului DP, pornind de la starea $DP(N, T)$. Un element a_{n-1} este inclus în soluție dacă starea $DP(n - 1, T - a_{n-1})$ este adevărată; în caz contrar, acesta este exclus. Procesul continuă până la atingerea sumei 0.

Metoda prezentată este una **exactă**, garantează identificarea unei soluții atunci când aceasta există și are complexitate **pseudo-polinomială**, dependența principală fiind de intervalul valorilor posibile ale sumelor și nu doar de dimensiunea mulțimii de intrare.

3.1.3 Genetic

Pentru rezolvarea problemei *Subset Sum* în prezența valorilor negative și a unui N mare, s-a ales o abordare euristică bazată pe **algoritmi genetici**. Această alegere este motivată de limitările metodelor exacte clasice, precum programarea dinamică, care devin impracticabile din punct de vedere al timpului și memoriei atunci când T este mare sau când sunt permise valori negative.

Algoritmul genetic operează pe o **populație de soluții candidate**, fiecare soluție fiind reprezentată printr-un cromozom binar de lungime N. O genă (element din cromozom) de valoare 1 semnifică faptul că elementul de pe indicele corespunzător este inclus în submultime, iar o genă de valoare 0 indică excluderea acestuia.

Calitatea fiecărei soluții este evaluată printr-o **funcție de fitness**, care urmărește maximizarea sumei elementelor selectate sub constrângerea $S \leq T$. Soluțiile care depășesc această constrângere sunt penalizate, ceea ce permite tratarea naturală a valorilor negative fără a impune mecanisme deterministe de corectare.

Procesul evolutiv este structurat pe **generații succesive**. La fiecare generație, indivizii sunt selectați pentru reproducere folosind o schemă de selecție de tip *turneu*, care favorizează soluțiile cu fitness mai ridicat. Din indivizii selectați se generează soluții noi prin aplicarea operatorului de **crossover uniform**, care combină genele a doi părinți pentru a produce un descendent.

Pentru a menține diversitatea populației și a evita convergența prematură către soluții locale, asupra descendenților se aplică un operator de **mutație**. Mutația constă în modificări locale ale cromozomului și este aplicată cu o probabilitate redusă. În implementarea de față, mutația este ușor direcționată în funcție de raportul dintre suma curentă și valoarea țintă, având rolul de a accelera convergența fără a compromite caracterul stocastic al metodei.

Pentru a asigura păstrarea celor mai bune soluții identificate, se utilizează o strategie de **elitism**, prin care un număr redus de indivizi cu fitness maxim sunt copiați nemodificat în generația următoare.

Algoritmul este de tip **anytime**, fiind oprit fie la atingerea unei soluții optime ($S = T$), fie la expirarea unui timp maxim de execuție. În orice moment, algoritmul returnează cea mai bună soluție fezabilă identificată până în acel moment.

Deși algoritmul genetic nu garantează obținerea soluției optime, acesta oferă un compromis eficient între calitatea soluției și timpul de execuție, fiind adecvat pentru instanțe mari sau pentru probleme extinse cu constrângeri suplimentare.

3.2 Analiza complexității

3.2.1 Backtracking

În cel mai nefavorabil caz, algoritmul explorează toate submulțimile posibile ale unei mulțimi cu N elemente. Arborele de decizie are câte două ramuri pentru fiecare element (*include / exclude*), deci numărul total de noduri este $\Theta(2^N)$.

Complexitatea în timp. În absența pruning-ului, complexitatea în timp este:

$$\boxed{\mathcal{O}(2^N)}.$$

În practică, timpul poate fi mai mic atunci când o soluție este găsită devreme, deoarece implementarea se oprește la prima soluție validă.

Complexitatea în spațiu. Spațiul auxiliar este dominat de adâncimea stivei de recursie (cel mult N apeluri) și de stocarea submulțimii curente (cel mult N elemente), deci:

$$\boxed{\mathcal{O}(N)}.$$

3.2.2 Dynamic Programming

Pentru analiza complexității, notăm cu N numărul de elemente din mulțimea de intrare și cu T valoarea țintă. Deoarece algoritmul permite valori negative, complexitatea nu depinde direct de T, ci de intervalul tuturor sumelor posibile.

Fie:

$$minSum = \sum_{a_i<0} a_i, \qquad maxSum = \sum_{a_i>0} a_i.$$

Domeniul tuturor sumelor realizabile este:

$$R = maxSum - minSum + 1.$$

Complexitatea în spațiu. Structura de date dominantă este tabelul de programare dinamică

$$memo \in \mathcal{M}_{N\times(R+1)}(\mathbb{Z})$$

Prin urmare, complexitatea spațială este:

$$\boxed{\mathcal{O}(N \cdot R)}.$$

Spațiul auxiliar utilizat pentru stocarea soluției reconstruite este $\mathcal{O}(N)$, care nu modifică ordinul dominant al consumului de memorie.

Complexitatea în timp. Fiecare stare $DP(n, S)$ este evaluată cel mult o singură dată datorită memoizării. Pentru fiecare astfel de stare se efectuează un număr constant de operații (două apeluri recursive și verificări de indici), ceea ce conduce la un cost constant per stare.

Numărul total de stări este:

$$(N + 1) \cdot R.$$

Prin urmare, complexitatea totală în timp pentru determinarea existenței unei soluții este:

$$\boxed{\mathcal{O}(N \cdot R)}.$$

Reconstrucția soluției. Etapa de reconstrucție a subsetului implică o parcurgere descendentă a cel mult N niveluri, fiecare pas efectuând verificări constante în tabelul DP. Complexitatea acestei etape este:

$$\mathcal{O}(N).$$

Aceasta este neglijabilă comparativ cu faza de calcul a tabelului DP.

Observație. Algoritmul are o complexitate **pseudo-polinomială**, deoarece depinde de mărimea numerică a valorilor din intrare prin intermediul domeniului R, și nu doar de dimensiunea N a instanței. În cazul în care valorile numerice sunt mari, consumul de memorie și timp poate deveni prohibitiv, limitare care motivează utilizarea unor metode euristice pentru instanțe extinse.

3.2.3 Genetic

Considerăm o populație de dimensiune P (în implementare: `POP`) și o lungime a cromozomului egală cu N (numărul de elemente). Fiecare individ este reprezentat printr-un vector binar de lungime N.

Complexitatea în spațiu. Structura dominantă în memorie este populația, care stochează P cromozomi, fiecare de lungime N. Prin urmare, spațiul necesar pentru populație este:

$$\mathcal{O}(P \cdot N).$$

Pe lângă populație, se mai stochează vectori auxiliari de dimensiune $\mathcal{O}(P)$ (de exemplu `fit`, `sumv`, vectorul de ordine) și datele de intrare $\mathcal{O}(N)$ (vectorul valorilor). Acestea nu modifică ordinul dominant, astfel încât:

$$\boxed{\text{Spațiu} = \mathcal{O}(P \cdot N)}.$$

Complexitatea în timp pe generație. Într-o generație sunt executate următoarele operații principale:

- **Sortarea populației după fitness.** Se sortează un vector de dimensiune P, deci:
$$\mathcal{O}(P \log P).$$
- **Generarea descendenților.** Se generează aproximativ $P - E$ indivizi noi (unde E este numărul de indivizi elite). Pentru fiecare descendent:
  - selecția părinților prin turneu are cost constant ($k = 3$), deci $\mathcal{O}(1)$;
  - *uniform crossover* parcurge toate cele N gene, deci $\mathcal{O}(N)$;
  - calculul sumei subsetului (`sum_of`) parcurge toate cele N gene, deci $\mathcal{O}(N)$;
  - mutația are un număr constant de încercări (de exemplu 25), deci $\mathcal{O}(1)$.

  Prin urmare, costul dominant per descendent este $\mathcal{O}(N)$, iar pentru P descendenți obținem:
$$\mathcal{O}(P \cdot N).$$

  Prin combinarea termenilor, complexitatea pe generație este:

$$\boxed{\text{Timp/generație} = \mathcal{O}(P \cdot N + P \log P)}.$$

În practică, pentru valori uzuale P (de ordinul sutelor), termenul $\mathcal{O}(P \cdot N)$ domină.

Complexitatea în timp total. Dacă algoritmul rulează pentru G generații, complexitatea totală este:

$$\boxed{\text{Timp total} = \mathcal{O}(G \cdot (P \cdot N + P \log P))}.$$

În varianta *anytime*, în care algoritmul se oprește după un timp maxim L (secunde), numărul efectiv de generații G devine dependent de timpul limită și de costul per generație. În acest caz, algoritmul returnează cea mai bună soluție găsită până la expirarea timpului, fără garanția atingerii soluției optime.

Observație practică. Implementarea efectuează atât crossover, cât și recalcularea sumei subsetului printr-o parcurgere completă a cromozomului. Astfel, pentru fiecare descendent se efectuează aproximativ două treceri peste N gene, ceea ce păstrează ordinul $\mathcal{O}(N)$, dar influențează factorii constanți. O optimizare posibilă ar fi calcularea sumei direct în timpul crossover-ului pentru a evita o trecere suplimentară.

Prin Table 1 se observă diferența fundamentală dintre metodele exacte (backtracking, DP) și abordarea euristică (GA): backtracking are timp exponențial în N, DP este pseudo-polinomial în funcție de intervalul R, iar GA depinde de parametrii algoritmului (dimensiunea populației P și numărul de generații G).

Table 1: Comparație a complexităților pentru algoritmii considerați.

Algoritm	**Timp (worst-case)**	**Spațiu**
Backtracking	$\mathcal{O}(2^N)$	$\mathcal{O}(N)$
Dynamic Programming (cu offset)	$\mathcal{O}(N \cdot R)$	$\mathcal{O}(N \cdot R)$
Genetic (GA)	$\mathcal{O}(G \cdot (P \cdot N + P \log P))$	$\mathcal{O}(P \cdot N)$

3.3 Avantaje și dezavantaje

3.3.1 Backtracking

Avantaje

- **Soluție exactă.** Dacă există o submulțime cu sumă T, metoda o va găsi.
- **Implementare simplă.** Algoritmul este ușor de implementat și de verificat.
- **Consum redus de memorie.** Spațiul auxiliar este liniar $\mathcal{O}(N)$.
- **Martor explicit.** Poate returna direct submulțimea soluție (nu doar răspunsul da/nu).

Dezavantaje

- **Timp exponențial în cel mai rău caz.** Complexitatea $\mathcal{O}(2^N)$ îl face nepractic pentru instanțe mari.
- **Sensibilitate la instanțe adversariale.** Există instanțe unde soluția apare foarte târziu sau nu există, forțând explorarea aproape completă a arborelui.
- **Fără garanții de timp.** Chiar pentru aceleași dimensiuni N, timpul poate varia semnificativ în funcție de distribuția valorilor și de poziția soluției în arbore.

Observație. În practică, performanța poate fi îmbunătățită prin *pruning* (de ex. oprirea unei ramuri când suma curentă depășește T în cazul numerelor nenegative, ordonarea descrescătoare a elementelor, sau memoizarea stărilor (i, rem)), însă aceste optimizări nu elimină caracterul exponențial în cazul general.

3.3.2 Dynamic Programming

Abordarea bazată pe programare dinamică oferă o soluție deterministă și exactă pentru problema *Subset Sum*, fiind una dintre cele mai utilizate metode pentru instanțele cu valori numerice moderate. Totuși, caracteristicile sale impun atât avantaje semnificative, cât și limitări importante în utilizarea practică.

Avantaje

- **Garanția corectitudinii.** Algoritmul de programare dinamică este o metodă exactă: dacă există o submulțime cu suma exact egală cu valoarea țintă, aceasta va fi identificată în mod cert.

- **Complexitate controlată pentru valori mici ale domeniului.** Atunci când intervalul tuturor sumelor posibile R este redus, algoritmul rulează eficient, având timp de execuție și consum de memorie rezonabile.
- **Reconstrucția explicită a soluției.** Spre deosebire de unele variante optimizate (de exemplu bitset), această abordare permite obținerea directă a submulțimii soluție, nu doar verificarea existenței acesteia.
- **Suport pentru valori negative.** Prin utilizarea unui offset, algoritmul poate trata valori negative fără a apela la transformări euristice sau la restructurări complexe ale problemei.
- **Comportament determinist și reproductibil.** Pentru aceleași date de intrare, algoritmul produce întotdeauna același rezultat, aspect important în aplicații critice sau în evaluări experimentale comparative.

Dezavantaje

- **Consum ridicat de memorie.** Dimensiunea tabelului de programare dinamică este $\mathcal{O}(N \cdot R)$, ceea ce poate deveni prohibitiv atunci când valorile numerice sunt mari, chiar și pentru valori moderate ale lui N.
- **Complexitate pseudo-polinomială.** Deși polinomial în N și R, algoritmul nu este polinomial în dimensiunea intrării binare, ceea ce reflectă caracterul NP-Complete al problemei Subset Sum.
- **Scalabilitate limitată.** Pentru instanțe cu domenii mari ale sumelor sau cu valori numerice dispersate, algoritmul devine impracticabil atât din punct de vedere al timpului de execuție, cât și al memoriei.
- **Inadaptabilitate la constrângeri suplimentare.** Extinderea metodei pentru a include criterii suplimentare (de exemplu optimizare multi-obiectivă) este dificilă comparativ cu abordările euristice.
- **Cost inutil pentru instanțe fără soluție.** Algoritmul parcurge întregul spațiu de stări chiar și atunci când nu există nicio submulțime validă, ceea ce poate conduce la timp de execuție ridicat fără un rezultat util.

3.3.3 Genetic

Abordarea prin algoritmi genetici prezintă o serie de avantaje și dezavantaje care trebuie luate în considerare în raport cu metodele deterministe clasice.

Avantaje

- **Scalabilitate pentru instanțe mari.** Algoritmul genetic nu depinde direct de valoarea sumei țintă T, spre deosebire de programarea dinamică pseudo-polinomială. Astfel, metoda rămâne aplicabilă pentru valori mari ale lui T și pentru un număr mare de elemente N.
- **Suport natural pentru valori negative.** Prin utilizarea unei funcții de fitness penalizate, algoritmul poate gestiona valori negative fără a necesita transformări suplimentare sau creșteri semnificative ale consumului de memorie.

– **Caracter anytime.** Algoritmul poate fi oprit în orice moment și returnează cea mai bună soluție fezabilă identificată până atunci, fiind potrivit pentru aplicații cu limită strictă de timp.
– **Flexibilitate ridicată.** Metoda permite integrarea ușoară a unor constrângeri sau obiective suplimentare (de exemplu, penalizări, ponderi sau criterii multiple), prin modificarea funcției de fitness.

Dezavantaje

– **Lipsa garanțiilor de optimalitate.** Algoritmul genetic nu garantează găsirea soluției optime, chiar dacă aceasta există, ci doar a unei soluții apropiate de optim.
– **Caracter nedeterminist.** Rezultatele pot varia între execuții diferite, datorită utilizării selecției aleatorii, crossover-ului și mutației.
– **Necesitatea ajustării parametrilor.** Performanța algoritmului depinde de alegerea parametrilor precum dimensiunea populației, rata de mutație sau dimensiunea turneului, care pot necesita calibrare empirică.
– **Cost computațional ridicat per soluție evaluată.** Fiecare evaluare a unui individ implică parcurgerea întregului cromozom, ceea ce poate conduce la un cost semnificativ pentru valori mari ale lui N.

Table 2: Avantaje și dezavantaje ale abordărilor analizate pentru *Subset Sum*.

Algoritm	**Avantaje**	**Dezavantaje**
Backtracking	Exact; implementare simplă; spațiu redus $\mathcal{O}(N)$; produce martorul soluției.	Timp exponențial $\mathcal{O}(2^N)$; instabil pe instanțe adversariale; fără garanții de timp în practică.
Dynamic Programming (cu offset)	Exact; timp controlat când R este moderat; reconstrucția soluției; suportă valori negative; determinist și reproductibil.	Consum mare de memorie $\mathcal{O}(N \cdot R)$; pseudo-polinomial; scalabilitate limitată când R este mare.
Genetic (GA)	Scalabil pentru N mare; nu depinde direct de T; suport natural pentru valori negative; anytime; flexibil pentru constrângeri suplimentare.	Nu garantează optimalitatea; nedeterminist; necesită calibrare parametri; cost per generație $\mathcal{O}(P \cdot N)$ poate fi ridicat.

Table 2 sintetizează compromisurile principale: metodele exacte oferă garanții de corectitudine, dar pot deveni impracticabile pentru instanțe mari, în timp ce abordarea genetică oferă rezultate rapide și robuste pentru dimensiuni mari, însă fără garanții de optimalitate.

4 Evaluare

4.1 Construirea setului de teste

Pentru validarea algoritmilor, s-a generat un set de teste sintetic, cu diverse dimensiuni și valori, folosind generatorul randomizat implementat în Python. S-a utilizat `random.seed(67)` pentru reproductibilitate.

Structura testelor. Fiecare test este stocat într-un fișier text de forma:

```
n T
a_1 a_2 ... a_n
```

unde:

- n reprezintă numărul de elemente din vector;
- T este suma țintă pentru problema Subset Sum;
- $a_1, a_2, \ldots, a_n$ sunt elementele vectorului, generate aleator în intervalul $[lo, hi]$ specific fiecărui test.

Clase de teste.

1. **Teste de dimensiuni mici și medii:** Instanțe cu $n = 10, 20, 30, 100$ și valori în intervale mici sau moderate pentru a evalua comportamentul algoritmilor în cazuri rapide.
2. **Teste mari:** Instanțe cu $n = 500, 1000, 2000, 2500, 5000$ și valori mai mari, pentru a analiza scalabilitatea și consumul de memorie al algoritmilor, în special al implementării de tip programare dinamică.
3. **Teste speciale / worst-case pentru backtracking:** În aceste instanțe suma țintă T este egală cu suma tuturor elementelor, forțând algoritmul să exploreze întreg spațiul de soluții și simulând cazul de complexitate maximă pentru backtracking. Exemple generate: `test_30_special`, `test_31_special`, `test_32_special`.
4. **Teste speciale / worst-case pentru programarea dinamică:** Aceste instanțe sunt construite astfel încât intervalul total al sumelor posibile

   $$[minSum,\ maxSum]$$

   să fie foarte larg. Se folosesc valori pozitive și negative de magnitudine mare, ceea ce conduce la un *range* ridicat al sumelor și, implicit, la un consum mare de timp și memorie pentru abordarea DP cu offset. Suma țintă T este nenulă, pentru a evita soluția trivială reprezentată de submulțimea vidă. Exemple generate: `test_200_dp_bad_offset`, `test_1000_dp_bad_offset`, `test_2000_dp_bad_offset`.
5. **Teste speciale / worst-case pentru abordarea genetică:** Aceste instanțe sunt concepute pentru a pune în dificultate algoritmul genetic. Suma țintă T este mică, iar doar un număr redus de elemente au valori suficient de mici pentru a contribui la o soluție fezabilă. Majoritatea elementelor au valori mari, care conduc rapid la depășirea lui T și la penalizarea fitness-ului. Această structură reduce semnalul evolutiv și îngreunează convergența către soluția optimă. Exemple generate: `test_200_ga_bad`, `test_300_ga_bad`, `test_1000_ga_bad`.

Strategia de generare a sumei țintă. Pentru testele normale, suma țintă T este aleasă aleator între 1 și $\min(\text{abs}(\text{sum}(\text{arr}))/2, 10^6)$, astfel încât:

- să fie fezabilă pentru algoritmi;
- să varieze nivelul de dificultate al instanțelor.

Pentru testele worst-case pentru backtracking, $T = \sum a_i$, forțând algoritmii deterministici să parcurgă toate combinațiile posibile înainte de a identifica soluția. Aici se poate observa că timpul de execuție pentru algoritmul backtracking se dublează pentru fiecare element în plus (mai multe informații la rezultate experimentale).

Pentru testele worst-ish-case pentru programarea dinamică, suma țintă T este aleasă nenulă (de exemplu $T = 3$), pentru a evita soluția trivială reprezentată de submulțimea vidă (care are sumă 0). În același timp, lista de valori este construită cu elemente pozitive și negative de magnitudine mare, ceea ce produce un interval foarte larg al sumelor posibile $[minSum, maxSum]$. Astfel, dimensiunea tabelului DP și numărul potențial de stări vizitate depind de $range = maxSum - minSum + 1$, iar pentru valori ale lui T situate în interiorul acestui interval, algoritmul poate ajunge să exploreze o parte semnificativă din spațiul $(n, range)$ înainte de a confirma existența unei soluții. În aceste instanțe se observă creșterea rapidă a timpului și a memoriei necesare odată cu mărirea lui *range*, confirmând caracterul pseudo-polinomial al metodei (mai multe detalii în rezultate experimentale).

Pentru testele worst-ish-case dedicate abordării genetice, suma țintă T este menținută foarte mică, iar doar un număr redus de elemente pot contribui la o soluție fezabilă. Majoritatea elementelor au valori mari, ceea ce determină depășirea rapidă a lui T. Algoritmul genetic continuă să furnizeze soluții aproximative îndepărtate într-un timp limitat, evidențiind caracterul său euristic și *anytime* (mai multe detalii în rezultate experimentale).

Această strategie permite evaluarea:

- corectitudinii algoritmilor pe instanțe ușoare și medii;
- performanței și scalabilității pe instanțe mari;
- comportamentului în cazuri extrem de dificile pentru backtracking.
- comportamentului în cazuri extrem de dificile pentru DP.
- comportamentului în cazuri extrem de dificile pentru algoritmul genetic.

4.2 Specificații sistem

Evaluările experimentale au fost efectuate pe următorul sistem:

- **Procesor:** AMD Ryzen 5 5600H (3.30 GHz, 6 nuclee)
- **Memorie RAM:** 32 GB (27.9 GB utilizabili)
- **Stocare:** 2.28 TB
- **Sistem de operare:** Ubuntu 22.04.3 LTS (WSL 2, Linux kernel 6.6.87.2 x86_64)
- **WSL version:** 2.6.3.0
- **Compilator / limbaj:** g++ 11.4.0

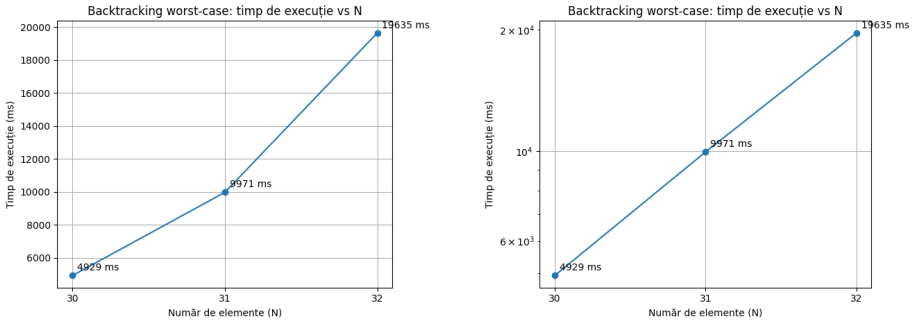
4.3 Rezultate experimentale

4.3.1 Comparație globală a timpilor de execuție Tabelul 3 prezintă timpii de execuție pentru câteva instanțe reprezentative. În cazul depășirii limitei de timp (30s), rezultatul este marcat ca *timeout* (5s in cazul algoritmului genetic).

Table 3: Timp de execuție (ms) pentru instanțe reprezentative

Test	Backtracking	DP (offset)	Genetic
test_30_small	∼100	∼100	∼100
test_100_medium	∼100	∼100	∼100
test_500_large	∼100	∼600	∼100
test_1000_large	∼100	∼14000	∼100
test_2000_large	∼100	timeout	timeout
test_2500_large	timeout	∼29700	timeout
test_5000_xlarge	∼12400	timeout	∼3800
test_30_special	∼5000	∼100	∼100
test_200_dp_bad_offset	timeout	∼25200	∼100
test_1000_ga_bad	timeout	∼5600	timeout

4.3.2 Evoluția timpului de execuție pentru backtracking Figura 1 ilustrează creșterea timpului de execuție al algoritmului de backtracking în funcție de numărul de elemente N, pentru testele de tip *worst-case*.



(a) Scară liniară (b) Scară logaritmică

Fig. 1: Timp de execuție pentru backtracking pe teste worst-case (`test_30_special`, `test_31_special`, `test_32_special`)

4.3.3 Impactul intervalului de sume asupra DP-ului cu offset Tabelul 4 conține rezultatele rulării algoritmului dinamic pe o serie de teste construite special pentru a stresa limitele programului. Elementul big reprezintă valoarea numerelor mari din test, iar range reprezintă diferența dintre cea mai mare și cea mai mică sumă posibilă, conform discuției de la construirea setului de teste.

Table 4: Performanța programării dinamice cu offset

Test	N	**big**	**range**	**Timp (ms)**
test_200_dp_bad_offset	200	100000	$\sim 2 \cdot 10^7$	~ 25200
test_1000_dp_bad_offset	1000	2000	$\sim 2 \cdot 10^6$	~ 11000
test_2000_dp_bad_offset	2000	2000	$\sim 4 \cdot 10^6$	timeout

4.3.4 Rezultatele execuției algoritmului genetic În tabelul 5 putem observa capacitatea programului care implementează abordarea genetică în raport cu dimensiunea testelor. S-a notat cu P procentajul de instanțe în care algoritmul a ajuns la un răspuns corect și cu $\overline{\mathbf{\Delta}}$ media diferențelor între suma căutată T și suma subsetului aproximativ găsit într-o execuție.

Table 5: Masuratori rulare

Test	P	$\overline{\mathbf{\Delta}}$	T
test_5000_xlarge	3%	13.63	362930
test_1000_large	57%	1.22	18068
test_100_medium	100%	0	347

De asemenea, testele special concepute pentru a evidenția slăbiciunile acestei abordări prezintă rezultate incredibil de îndepărtate de răspunsul corect, fapt evidențiat in Tabelul 6.

Table 6: Sumele subseturilor gasite

Test	**Sum**	T
test_1000_ga_bad	456001	3
test_300_ga_bad	78040	100
test_200_ga_bad	79005	10

4.4 Interpretarea rezultatelor

Rezultatele experimentale evidențiază diferențe clare între cele trei abordări analizate, atât din punct de vedere al complexității, cât și al comportamentului practic pe instanțe nefavorabile.

Algoritmul de backtracking prezintă o creștere exponențială a timpului de execuție în testele de tip *worst-case*, unde suma țintă T este egală cu suma tuturor elementelor. Chiar și o creștere modestă a lui N conduce la dublarea timpului de execuție, ceea ce face metoda impracticabilă pentru instanțe mai mari.

Programarea dinamică cu offset oferă performanțe foarte bune pe instanțe generice, dar devine rapid limitată de consumul de memorie și timp atunci când intervalul sumelor posibile este mare. Testele special construite arată că, deși N rămâne moderat, creșterea lui *range* conduce la timeout-uri sau la timpi de ordinul zecilor de secunde, confirmând caracterul pseudo-polinomial al metodei.

Abordarea genetică prezintă un comportament robust pe toate categoriile de teste, fără a depăși limita de timp. Deși nu garantează găsirea soluției optime și produce adesea subseturi mai mari decât cele obținute prin metode deterministe, algoritmul genetic reușește să furnizeze soluții fezabile într-un timp aproape constant, independent de structura nefavorabilă a instanței. Acest lucru evidențiază avantajul metodelor euristice în contexte în care abordările deterministe devin impracticabile.

5 Concluzii

În această lucrare a fost analizată problema *Subset Sum* folosind trei abordări distincte: backtracking, programare dinamică cu offset pentru numere negative și algoritmi genetici. Analiza teoretică a fost susținută de o serie de experimente controlate, menite să evidențieze comportamentul fiecărui algoritm în condiții favorabile și nefavorabile.

Algoritmul de tip backtracking oferă soluții exacte, însă suferă de o creștere exponențială a timpului de execuție. Testele de tip worst-case, în care suma țintă este egală cu suma tuturor elementelor, au confirmat experimental complexitatea de ordin $O(2^N)$, evidențiind caracterul impracticabil al acestei abordări chiar și pentru valori moderate ale lui N.

Programarea dinamică cu offset extinde aplicabilitatea metodei clasice de Subset Sum la instanțe care conțin valori negative. Totuși, performanța acestei abordări este dominată de intervalul total al sumelor posibile, $range = maxSum - minSum + 1$, și nu de numărul de elemente. Experimentele au arătat că pentru valori mari ale acestui interval, consumul de memorie și timpul de execuție devin prohibitive, limitând utilizarea practică a metodei la instanțe cu amplitudine numerică redusă.

Algoritmul genetic, deși nu garantează obținerea unei soluții exacte, a demonstrat un comportament robust pe instanțe de dimensiuni mari. Rezultatele experimentale indică faptul că, pe măsură ce dimensiunea problemei crește, probabilitatea de a găsi soluția exactă scade, însă deviația medie față de suma țintă rămâne redusă. Această proprietate confirmă caracterul euristic și *anytime* al abordării genetice, care poate furniza soluții aproximative de bună calitate într-un timp controlabil.

În concluzie, nu există o soluție universal optimă pentru problema Subset Sum. Alegerea metodei depinde de natura instanței: backtracking este adecvat pentru dimensiuni mici, programarea dinamică este eficientă atunci când intervalul numeric este restrâns, iar algoritmii genetici reprezintă o alternativă practică pentru instanțe foarte mari, unde soluțiile aproximative sunt acceptabile.

References

1. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman (1979)
2. Horowitz, E., Sahni, S.: Computing partitions with applications to the knapsack problem. *Journal of the ACM* **21**(2), 277–292 (1974)
3. Karp, R. M.: Reducibility Among Combinatorial Problems. In: Miller, R. E., Thatcher, J. W. (eds.) *Complexity of Computer Computations*, pp. 85–103. Plenum Press (1972).
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: *Introduction to Algorithms*. 3rd edn. MIT Press (2009).
5. Bellman, R.: Dynamic Programming. Princeton University Press (1957).
6. Goldberg, D. E.: *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley (1989).
7. Mitchell, M.: *An Introduction to Genetic Algorithms*. MIT Press (1998).
8. Eiben, A. E., Smith, J. E.: *Introduction to Evolutionary Computing*. Springer (2003).