Problema Subset Sum

Nicolescu Petre and Oprea Andrei-Catalin
324CD

Facultatea de Automatica si Calculatoare
Universitatea POLITEHNICA Bucuresti

1 Introducere

1.1 Problema propusă

Problema **Subset Sum** este o problemă fundamentală în teoria complexității și algoritmicii. Cerința problemei oferă o mulțime finită de obiecte, fiecare având asociată o valoare întreagă, și un număr întreg țintă. Obiectivul este de a identifica o submulțime astfel încât suma valorilor obiectelor din submulțimea selectată să fie egală cu ținta. Dacă există mai multe submulțimi care satisfac această condiție, orice astfel de submulțime este considerată o soluție validă.

În ciuda simplității acestei probleme, este **NP-completă**, ceea ce înseamnă că nu există un algoritm eficient care să rezolve întotdeauna această problemă într-un timp polinomial, pentru toate instanțele posibile. Astfel, soluțiile exact corecte necesită adesea o cantitate semnificativă de resurse de calcul, în special pentru instanțe mari ale problemei. În general, există un compromis între complexitatea algoritmică (timpul de execuție și spațiul necesar) și precizia soluției.

Există numeroase tehnici și algoritmi care abordează această problemă: Soluțiile exacte, care garantează identificarea precisă a unui rezultat, au complexități ridicate, în timp ce abordările inexacte oferă eficiență temporală, dar rezultate doar aproximative.

1.2 Aplicații practice

Subset Sum nu se regăsește doar în domenii strict teoretice, ci are numeroase utilizări practice în domenii actuale.

În domeniul financiar, problema optimizării investițiilor poate fi formulată (și eventual rezolvată) sub o formă asemănătoare cerinței noastre: se urmărește selecția unui subset de investiții care să maximizeze profitul, respectând o sumă maximă de capital disponibil.

Putem găsi aplicații ale problemei chiar în domeniul criptografiei/securității: Concepte precum parole sau mesaje criptate pot fi implementate cu ajutorul unor submulțimi alese după formatul propus de Subset Sum.

Importanța acestei probleme devine clară mai ales când vine vorba de domeniul educației. Datorită simplității sale, atât problema, cât și algoritmii care o rezolvă devin exemple introductive perfecte pentru conceptul de probleme NP-complete.

2 Demonstrație NP-Hard

Teorema: O problemă P este considerată **NP-Hard** dacă, pentru orice altă problemă din clasa NP (non-deterministic polinomial) există o reducere polinomială de la acea problemă la problema P.

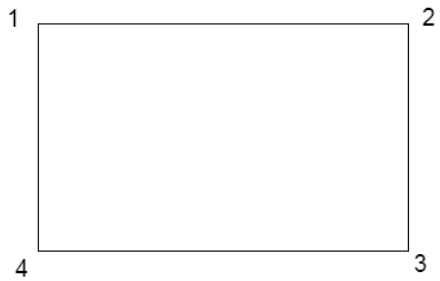
Pentru a demonstra faptul că problema **Subset Sum** este NP-Hard, putem alege o altă problemă despre care deja știm că este NP-Hard și să o reducem la problema noastră. În cadrul acestei secțiuni, vom opta pentru **Vertex Cover**, care oferă un număr natural K și un graf neorientat $G(V, E)$, unde V este mulțimea vârfurilor și E este mulțimea muchiilor, iar apoi propune găsirea unei submulțimi de K vârfuri notată S astfel încât fiecare muchie a grafului să aibă cel puțin un capăt în submulțimea S.

Pentru această reducere, trebuie să asociem datele de intrare ale problemei Subset Sum cu Vertex Cover prin intermediul unor valori naturale scrise în baza 4 și cu $|E| + 1$ cifre, care vor fi plasate drept linii în interiorul unei matrice:

- a_i va fi asociat vârfului i din graf, unde cifra j este 1 dacă muchia e_j este incidentă în vârful i, și 0 altfel (cu excepția cifrei cea mai semnificative, care va fi pentru toate aceste numere 1).
- b_{ij} va fi asociat muchiei e_{ij} din graf, cu prima cifră de data asta 0 și cu o singură cifră de 1, pe coloana corespunzătoare muchiei pe care o reprezintă.

Aceste valori pot părea destul de abstracte, așa că este benefic să le vizualizăm prin intermediul unui exemplu:

Exemplu Fie un graf $G(V, E)$ format din vârfurile $V = \{1, 2, 3, 4\}$ și muchiile $E = \{(1, 2); (2, 3); (3, 4); (4, 1)\}$, iar $K = 2$ va fi numărul de vârfuri al subgrafului soluție:



Pentru această figură, construim o matrice folosind valorile enunțate mai devreme; valorile de tip a_i se asociază vârfurilor, cele de tip b_{ij} muchiilor, iar e_{ij} reprezintă chiar coloanele asociate fiecărei muchii:

	e_{12}	e_{23}	e_{34}	e_{14}
a_1	1	1	0	0	1
a_2	1	1	1	0	0
a_3	1	0	1	1	0
a_4	1	0	0	1	1
b_{12}	0	1	0	0	0
b_{23}	0	0	1	0	0
b_{34}	0	0	0	1	0
b_{14}	0	0	0	0	1

În cadrul demonstrației noastre, numerele de tip a și b reprezintă set-ul nostru de numere pentru Subset Sum. Pe lângă aceste valori, problema are nevoie și de suma țintă, pe care o definim astfel:

$$T = k(4^{|E|}) + \sum_{i=0}^{|E|-1} 2(4^i)$$

Această formulare a sumei țintă descrie de fapt tot un număr natural definit în baza 4 de forma $K22...2$, ales în mod specific pentru setul nostru de date (Observați că suma elementelor de pe prima coloană este k, iar alegerea unei anumite submulțimi de valori poate determina ca restul coloanelor să aibă suma elementelor fix 2!).

Să alegem un submulțimile V' și E', cu următoarea proprietate:

$$\sum_{i \in V'} a_i + \sum_{(i,j) \in E'} b_{ij} = T$$

Știm că pentru suma elementelor b_{ij} vom avea maxim 1 pe fiecare coloană (în afară de prima), care după adunarea cu suma elementelor a_i trebuie să devină 2, așa cum impune forma lui T. Deoarece am scris aceste numere în baza 4 și fiecare muchie are strict 2 vârfuri, suma pe o singură coloană nu va trece niciodată de 3 și nu vom avea nici un 'carry' care poate interveni cu suma unei alte coloane. Luând în considerare aceste detalii, implicația devine clară: mulțimea V' trebuie să conțină, pentru fiecare muchie inclusă de b_{ij}, vârful i sau j, ceea ce reprezintă chiar o submulțime soluție pentru **Vertex Cover**. Astfel, putem considera numerele ce formează V' și E' elementele subset-ului căutat de **Subset Sum**, care are drept sumă chiar valoarea țintă T.

Să considerăm exemplul de mai devreme, cu $V' = \{1, 3\}$, $K = 2$ un Vertex Cover (și, implicit, $E' = E$). Valoarea lui T devine 682, iar apoi alegem a_i și b_{ij} conform Vertex Cover-ului dat:

- $a_1 = 11001_4 = 321$

- $a_3 = 10110_4 = 276$
- $b_{12} = 01000_4 = 64$
- $b_{23} = 00100_4 = 16$
- $b_{34} = 00010_4 = 4$
- $b_{41} = 00001_4 = 1$

Se observă $321 + 276 + 64 + 16 + 4 + 1 = 682$, ceea ce înseamnă că aceasta este chiar soluția pentru **Subset Sum**.

Rezultat: Am reușit să reducem problema **Vertex Cover** la problema **Subset Sum**, ceea ce înseamnă că putem să o considerăm **NP-Hard**.

3 Algoritmi

3.1 Backtracking

Algoritmul **Backtracking** este o abordare de căutare exhaustivă, care explorează toate combinațiile posibile de obiecte din mulțimea dată pentru a verifica dacă suma lor este egală cu 'target sum'. Pentru fiecare valoare, propunem două opțiuni: includerea și excluderea sa din submulțimea finală. Calculăm suma actuală a submulțimii, iar în cazul în care aceasta depășește suma căutată, abandonăm ramura curentă și ne întoarcem cu un pas în spate (backtrack).

Acest procedeu poate fi asemănat cu un arbore binar, mai ales având în vedere complexitatea acestui algoritm, de $O(2^n)$, unde n este dimensiunea mulțimii date. Fiind exponențială, complexitatea reprezintă unul din cele mai mari dezavantaje ale acestui algoritm. Un bun exemplu l-am descoperit în timp ce generam teste: în cazurile în care nu există nici o submulțime care să îndeplinească cerința, pentru doar 30 de valori, timpul de execuție a ajuns la 5 secunde, dublându-se pentru fiecare valoare adăugată în mulțime. Datorită apelurilor recursive, algoritmul ocupă și mult spațiu pe stivă, iar de multe ori se obțin sume deja calculate de-a lungul procesului, care devin redundante. În ciuda acestor atribute negative, algoritmul **Backtracking** este una dintre cele două abordări care obțin o submulțime exactă, indiferent de datele de intrare.

Observație! În cazul testelor care conțin valori negative, pentru o validare corectă a sumelor, este necesară eliminarea condiției de oprire ($sum > target$), din moment ce avem și numere negative care pot afecta sume mari. Datorită acestei schimbări, acest algoritm poate deveni și mai ineficient în unele cazuri!

```
1  bool backtrack(int *arr, int n, int target, int sum, int *subset,
2                 int size, int idx) {
3      if (sum == target) {
4          print_subset(subset, size);
5          return true;
```

```
    }
    if (sum > target)
        return false;
    if (idx == n)
        return false;
    subset[size] = arr[idx];
    return backtrack(arr, n, target, sum + arr[idx], subset, size + 1, idx + 1) ||
           backtrack(arr, n, target, sum, subset, size, idx + 1);
}
```

3.2 Programare Dinamică

Acest algoritm implică construirea unei matrice cu $n + 1$ linii și $target + 1$ coloane, unde n reprezintă dimensiunea mulțimii date și $target$ este suma țintă. Elementele matricei au valori booleene, care determină dacă suma j poate fi obținută dintr-o submulțime a elementelor până la poziția i (i și j fiind coordonatele unui element $matrix[i][j]$ din matrice). Prima coloană este inițializată cu valori *true* (deoarece orice mulțime are o submulțime cu suma 0), iar restul elementelor sunt determinate printr-un procedeu vizual asemănător cu cel aplicat în cadrul primului algoritm prezentat. În schimb, observăm o variație a complexității (legându-ne în principal doar de construirea matricei), care este $O(n \times target)$.

Față de algoritmul **Backtracking**, această metodă are o complexitate mai favorabilă și nu suferă de aceleași limitări când vine vorba de natura datelor de input pentru obținerea acelorasi rezultate. În schimb, această abordare suferă din punct de vedere al utilizării memoriei, deoarece dimensiunea/alocarea matricei de valori depinde în mod direct atât de numărul de elemente, cât și de suma țintă.

```
void dynamic_subset(int *arr, int target, int n) {
    bool **dp = calloc((n + 1), sizeof(bool *));
    for (int i = 0; i <= n; ++i)
        dp[i] = calloc((target + 1), sizeof(bool));
    for (int i = 0; i <= n; ++i)
        dp[i][0] = true;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= target; ++j) {
            if (j < arr[i - 1])
                dp[i][j] = dp[i - 1][j];
            else
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j - arr[i - 1]];
        }
```

```
    }
    if (!dp[n][target]) {
        printf("Subset not found!\n");
        return;
    }
    int *subset = malloc(n * sizeof(int));
    int idx = 0, sum = target;
    for (int i = n; i > 0 && sum > 0; i--) {
        if (dp[i][sum] != dp[i - 1][sum]) {
            subset[idx++] = arr[i - 1];
            sum -= arr[i - 1];
        }
    }
    print_subset(subset, idx);
}
```

3.3 Greedy

Algoritmul **Greedy** pentru rezolvarea problemei **Subset Sum** nu garantează întotdeauna găsirea unei soluții exacte, ci se concentrează pe obținerea unui set de date al căreir sumă se apropie de valoarea țintă. Procedura începe cu un vector de elemente deja sortat în funcție de modulele numerelor pentru un rezultat optim. Această sortare poate fi realizată cu ajutorul unui algoritm de sortare eficient, cum ar fi **QuickSort**, care are o complexitate temporală de $O(n \times logn)$, unde **n** reprezintă numărul de elemente din subsetul inițial. După sortare, se iterează prin vectorul de valori, iar la fiecare pas, se evaluează euristic reducerea diferenței dintre suma curentă și suma țintă dacă elementul ar fi adăugat. Se adaugă elementul doar dacă acesta contribuie la micșorarea diferenței.

În ceea ce privește complexitatea temporală, ea este complexitatea **QuickSort** adunată complexității iterării și adăugării: $O(n \times logn + n)$, adică $O(n \times logn)$, unde n reprezintă numărul de elemente prezente în subsetul inițial. Pentru complexitatea spațială, nu se alocă decât o singură dată subsetul și o dată vectorul luat din fișierele de teste, deci este doar $O(n)$.

Greedy generează soluții optime locale, nu soluții globale optime. Astfel, în anumite cazuri, alegerea unui element poate duce la o soluție care este apropiată de obiectivul dorit, dar nu neapărat optimă, întrucât nu analizează toate posibilitățile. De exemplu, se pot include numere care, la o analiză globală, ar putea complica atingerea exactă a sumei dorite. Pe teste bine alese se poate ajunge chiar la soluția exactă cu un număr corespunzător de iterații.

```
int* greedy_ss(int* arr, int n, int* size, int err, int target) {
    int* subset = malloc(sizeof(int) * n);
    *size = 0;
```

```
    int diff, sum = 0;
    for(int i = 0; i < n; i++) {
        diff = abs_value(sum - target);
        if (diff <= err)
            break;
        if (abs_value(arr[i] + sum - target) < diff) {
            subset[*size] = arr[i];
            (*size)++;
            sum += arr[i];
        }
    }
    return subset;
}
```

3.4 Monte Carlo

Monte Carlo este un algoritm statistic iterativ care calculează sume alese aleatoriu cu o funcție de tip **Random** de un număr finit de ori, ales. De aceea, complexitatea este $O(n \times iter)$, unde iter reprezintă numărul de iterații iar n lungimea vectorului, care este parcurs o dată și în care se adună valorile aleatorii. El nu presupune alegerea sumei bazată pe optime locale, de aceea fiind mai ineficient în găsirea soluțiilor în anumite situații. Cu toate acestea, avantajul lui Monte Carlo este că explorează soluții pe care Greedy nu le-ar lua în considerare, din cauza ordinii fixe a elementelor. Față de Greedy, unde ordinea inițială a elementelor contează, acest algoritm este total aleator și poate ajunge la o soluție mai apropiată de adevăr. Totuși, **Monte Carlo** poate fi utilizat în combinație cu **Greedy** pentru a îmbunătăți performanța generală a algoritmilor, având rolul de a explora soluții alternative care ar putea fi mai bune decât cele obținute printr-o abordare strict **Greedy**.

```
int* montecarlo(int n, int* arr, int iterations, int target, int* size) {
    int* best_values = malloc(n * sizeof(int));
    int best_sum = 0;
    for (int i = 0; i < iterations; ++i) {
        int* curr_subset = malloc(n * sizeof(int));
        int curr_size = 0;
        int curr_sum = 0;
        for (int j = 0; j < n; ++j) {
            if (take_number_or_not()) {
                curr_subset[curr_size] = arr[j];
                curr_sum += arr[j];
                curr_size++;
```

```
            }
        }
        if (abs_value(curr_sum - target) < abs_value(best_sum - target)) {
            memcpy(best_values, curr_subset, curr_size * sizeof(int));
            *size = curr_size;
            best_sum = curr_sum;
        }
        free(curr_subset);
        if (curr_sum == target)
            break;
    }
    return best_values;
}
```

4 Evaluarea soluțiilor

4.1 Generarea soluțiilor

Pentru testele efectuate, au fost generate 7 tipuri de fișiere, fiecare cu caracteristici specifice cu ajutorul unui script Python modificat ușor pentru fiecare tip de test:

1. Fișiere cu numere mari pozitive, cu valori cuprinse între 10,000 și 100,000.
2. Fișiere cu numere mari, atât pozitive cât și negative, având intervalele de valori între 10,000 și 1,000,000 sau între -1,000,000 și -10,000. Aceste fișiere au fost alese pentru a evidenția diferențele de comportament între algoritmul de backtracking aplicat pentru numere negative și cel aplicat pentru numere pozitive.
3. Fișiere cu numere mici pozitive, având valori cuprinse între 10,000 și 100,000.
4. Fișiere cu numere mici, în intervalul [-1,000, 1,000], alese cu scopul de a adăuga diversitate față de cele cu numere mari.
5. Fișiere cu numere aleatoare, în intervalul [-1,000,000, 1,000,000].
6. Fișiere cu numere aleatoare pozitive, cu valori în intervalul [0, 1,000,000].
7. Fișiere cu soluție imposibilă (setul inițial de valori conține numai zerouri), utilizate pentru a evidenția complexitatea algoritmului de backtracking în condițiile în care nu există o soluție validă.

Menționăm că soluțiile căutate au fost alese în concordanță cu intervalele valorilor generate, astfel încât să fie relevante pentru tipologia datelor.

4.2 Specificațiile sistemului de calcul

Testele au fost rulate pe un sistem cu următoarele specificații tehnice:

– Sistem de operare: Microsoft Windows 11 Education

– Procesor: Processor 13th Gen Intel(R) Core(TM) i7-13650HX, 2600 Mhz, 14 Core(s), 20 Logical Processor(s)
– Memorie RAM disponibilă: SODIMM 12 GB RAM x 2

4.3 Evaluarea soluțiilor pe setul de teste

Vom începe prin a prezenta diferențele dintre **Monte Carlo** și **Greedy**. Am încercat să alegem numărul de iterații pentru **Monte Carlo** astfel încât timpii de execuție să fie aproximativ egali (400 iterații). După aceea, am rulat pe toate fișierele ambii algoritmi și am observat erorile obținute. Vom atașa rezultatele: timpul de execuție și output-ul rezultat: eroarea.

File	Executable	Execution Time (s)	Output
big_number_file0.txt	greedy	0.01181936264038086	48049
big_number_file0.txt	monte_carlo	0.018633604049682617	37710
big_number_file1.txt	greedy	0.01106119155883789	41596
big_number_file1.txt	monte_carlo	0.021660804748535156	17259
big_number_file10.txt	greedy	0.016731739044189453	53142
big_number_file10.txt	monte_carlo	0.01739192008972168	29302
big_number_file11.txt	greedy	0.015734195709228516	47503
big_number_file11.txt	monte_carlo	0.021648168563842773	40685
big_number_file12.txt	greedy	0.015437602996826172	45482
big_number_file12.txt	monte_carlo	0.0202481746673584	10567
big_number_file13.txt	greedy	0.01499485969543457	48684
big_number_file13.txt	monte_carlo	0.020834922790527344	33264
big_number_file14.txt	greedy	0.01662135124206543	50340
big_number_file14.txt	monte_carlo	0.0175325870513916	19519
big_number_file15.txt	greedy	0.015885591506958008	1813
big_number_file15.txt	monte_carlo	0.023253440856933594	1813

Fig. 1. Figura care ilustrează rezultatele algoritmilor Greedy și Monte Carlo pe fișiere cu numere mari.

Observăm că **Greedy** dă rezultate destul de slabe atunci când fișierele au numai numere, atât negative, cât și pozitive mari, dar în modul mai mari decât rezultatul căutat, deoarece nu ține cont de optime globale, iar cele locale nu îi oferă oportunități de micșorare de eroare. Cu toate acestea, la toate celelalte tipuri de fișiere (în afară de cele imposibile) **Greedy** întrece **Monte Carlo**, cu excepția câtorva cazuri.

small_number_file17.txt	greedy	0.007937192916870117	0
small_number_file17.txt	monte_carlo	0.024222612380981445	56
small_number_file18.txt	greedy	0.008821964263916016	0
small_number_file18.txt	monte_carlo	0.023584842681884766	165

Fig. 2. Figura care ilustrează găsirea soluției exacte de către Greedy, pe fișiere cu numere mici

Se observă că pentru numere mici, atât pozitive cât și negative, **Greedy** dă succese, având rezultate mai bune în **toate celelalte teste**, datorită faptului că există mai multe optime locale, de câteva ori găsindu-se chiar soluția exactă, așa cum se poate observa în imaginea de mai sus.

positive_file0.txt	greedy	0.007023334503173828	91193
positive_file0.txt	monte_carlo	0.01206517219543457	15275784
positive_file1.txt	greedy	0.0067183971405029 3	42780
positive_file1.txt	monte_carlo	0.00893402099609375	53003

Fig. 3. Figura care ilustrează o anomalie la Monte Carlo și puterea algoritmului Greedy care îl depășește în mod absolut

Explicația pentru această anomalie este că multe numere din acel fișier sunt mari, iar algoritmul le-a omis pe cele mici în toate iterațiile. Deci, dacă setul de date este foarte nepotrivit, **Greedy** nu va da o soluție anormală la fel ca **Monte Carlo**, dar nu ne putem baza nici pe rezultatul acestuia.

În graficele de mai jos am comparat erorile de la Monte Carlo cu numărul de iterații.

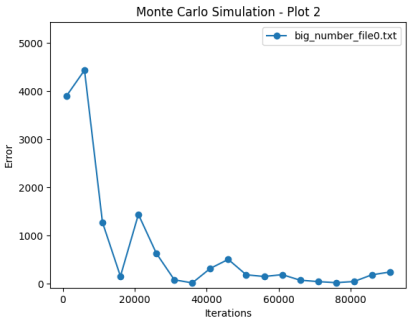


Fig. 4. Fișier cu numere mari

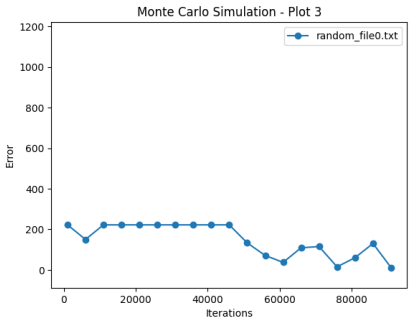


Fig. 5. Fișier cu numere de orice tip

Am avut 5 fișiere diferite pentru care am făcut graficele, dar celelalte 3 au avut rezultate aproape constante. Prin aceste 2 grafice vrem să evidențiem natura aleatorie a algoritmului, care cu creșterea numărului de iterații ar trebui în mod normal să scadă treptat, dar din cauze statistice observăm că uneori cu mai puține iterații vom obține rezultate mai bune.

Comparativ cu algoritmii care găsesc soluția exactă, cei care doar o aproximează sunt mult mai rapizi, mai puțin cel de programare dinamică, dar care nu poate fi folosit în orice situație. Spre exemplu, pentru a parcurge toate combinațiile fișierului imposibil, pe procesorul declarat mai sus a durat 9 secunde pentru 31 de numere pentru algoritmul de backtracking. În schimb, algoritmul de programare dinamică a rezolvat un fișier cu 1000 de numere și o sumă țintă de 100,000 în doar 0.5 secunde. Totuși, algoritmul de programare dinamică are o limitare majoră: nu funcționează corect în cazul în care inputul conține numere negative. În astfel de cazuri, o soluție ar fi filtrarea numerelor negative înainte de procesare, însă această abordare poate duce la pierderea multor combinații valide, ceea ce afectează calitatea rezultatelor.

Algoritmul de **Backtracking** pentru numere pozitive este mult mai rapid decât cel care funcționează pentru numere negative. Motivul este faptul că există condiția de oprire care simplifică mult recursivitatea algoritmului pe multe ramuri, scurtând timpul de execuție în mod considerabil. Dar, pe un fișier cu valori a cărei sumă este aleasă prost (spre exemplu fișierul imposibil), condiția de oprire nu are loc, deci algoritmul se comportă la fel ca cel neoptimizat, care are o complexitate de $O(2^n)$. Acest lucru este evidențiat în graficul de mai jos:

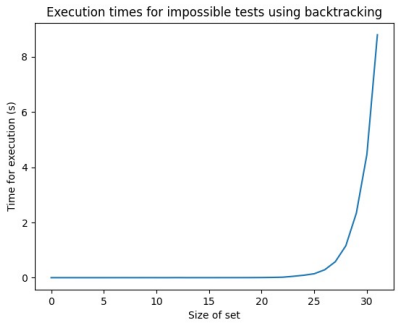


Fig. 6. Figura care ilustrează complexitatea exponențială a algoritmului de backtracking

Size of set reprezintă numărul de obiecte din setul de date primit, iar faptul că timpul de execuție crește logaritmic abia de la 23 în sus este că celelalte operații iau mai mult timp până la acel număr decât recursivitatea algoritmului.

Cu toate acestea, din păcate, algoritmul de programare dinamică nu poate fi folosit în toate situațiile:

1. dacă avem în input numere negative sau de tip float,
2. dacă dorim aproximare parțială -> este mai flexibil algoritmul de backtracking,
3. dacă suma țintei căutate este foarte mare, matricea alocată ar fi uriașă, și la fel și operațiile făcute pe ea ar dura mult,
4. dacă dorim mai mult decât o soluție ar fi foarte costisitor de implementat cu programare dinamică.

În continuare, comparăm diferențele dintre algoritmul de **Programare Dinamică** și **Backtracking-ul**:

big_number_positive_file0.txt	backtracking_positive	0.020740032196044922
big_number_positive_file0.txt	dp	0.028036832809448242
big_number_positive_file1.txt	backtracking_positive	0.013731718063354492
big_number_positive_file1.txt	dp	0.04080557823181152
big_number_positive_file10.txt	backtracking_positive	0.014813899993896484
big_number_positive_file10.txt	dp	0.05470418930053711

Fig. 7. Figura care ilustrează diferențele mici de timp de execuție pe testele făcute de noi între backtracking pe numere pozitive și dynamic programming

Datorită naturii testelor, care ajută la funcționarea corectă a ambilor algoritmi, am observat că timpii (în secunde) sunt asemănători pentru toate testele pozitive, nu doar cel arătat în imaginea de mai sus.

5 Concluzii

Fiecare algoritm are atât avantajele, cât și dezavantajele sale. Noi trebuie să profităm de avantaje în alegerea algoritmului pentru un anumit set de date primit. În general, dacă nu avem restricțiile dinamice specificate mai sus, algoritmul cu programare dinamică ar trebui ales. Dacă setul de date este pozitiv și suma nu prea mare pentru a nu exista complexitate de $O(2^n)$, backtrackingul pozitiv este exact ceea ce avem nevoie. În schimb, dacă setul de date are dimensiune mică

și elemente negative, putem folosi backtracking fără condiție de oprire fiindcă nu ne afectează complexitatea. Altfel, pentru a genera o soluție aproximativă, putem folosi Greedy și mai apoi itera cu Monte Carlo dacă nu ajungem la eroarea maximă dorită. În final, trebuie să fim foarte atenți la setul de date dat, deoarece există seturi de date la care aproximarea nu funcționează deloc, iar algoritmii de căutare exactă funcționează într-o durată prea mare sau nu funcționează, putând ajunge la problema opririi, deci nu ne putem încrede complet în niciunul dintre algoritmi.

6 Bibliografie

1. Subset Sum applications
2. Geeksforgeeks article on SubsetSum
3. Other Subset Sum uses
4. Greedy algorithms
5. NP-Hard Lecture