Tema 0 - Proiectarea Algoritmilor

Barus Catalin-Stefan, Grupa: 321CD

April 2021

1 Tehnica Divide et Impera

1.1 Enuntul problemei

Sa se sorteze crescator un vector, folosind un algoritm ce foloseste tehnica Divide et Impera.

1.2 Descrierea solutiei problemei

O solutie, ce poate rezolva aceasta problema, folosind tehnica de Divide et Impera este folosirea algoritmului QuickSort. Acest algoritm presupune alegerea unui pivot, dupa care tot vectorul se va partitiona. Alegerea unui pivot se poate face in mai multe feluri, acesta poate fi primul element din vector, ultimul, sau un element la intamplare. Odata ales, va trebui sa partionam vectorul astfel: toate elementele mai mici decat pivotul vor fi pozitionate in stanga lui, iar toate elementele mai mari in dreapta. Acest proces se va repeta recursiv, pentru toate partitiile posibile, cat timp avem elemente in vector. Pentru simplitate, solutia prezentata mai jos presupune tot timpul ca pivotul va fi ultimul element din vector.

1.3 Pseudocodul algoritmului

```
quickSort(v[], start, end)
{
    if (start < end)
    {

        partIndex = partition(v, start, end);

        quickSort(v, start, partIndex - 1);
        quickSort(v, partIndex + 1, end);
    }
}
```

```
/* Aceasta functie alege pivotul ca fiind ultimul element
   din vector, il pune in pozitia potrivita in vector, dupa
   care toate elementele mai mici decat el, vor fi puse in
   stanga pivotului, iar toate elementele mai mari, in dreapta
   pivotului. */

partition (v[], start, end)
{
    // pivot (Element to be placed at right position)
    pivot = v[end];

    i = (start - 1)  // Pozitia actuala a pivotului

    for (j = start; j <= end - 1; j++)
    {
        // Daca elementul e mai mic decat pivotul
        if (v[j] < pivot)
        {
            i++;    // incrementeaza indexul elementului mai mic
            swap v[i] & v[j]
        }
    }
    swap v[i + 1] & v[end])
    return (i + 1)
}
```

Pornind de la presupunerea ca pivotul va fi mereu ultimul element din vector, logica algoritmului umreaza niste pasi simpli: parcurgem de la stanga vectorul, iar daca gasim un element mai mic decat pivotul, facem swap intre elementul curent si pivot.

1.4 Complexitatea algoritmului

In general, complexitatea pentru QuickSort, poate fi reprezentata folosind urmatoarea recurenta:

$$\mathrm{T(n)} = \mathrm{T(k)} + \mathrm{T(n\text{-}k\text{-}1)} + \theta(n)$$

Primii 2 termeni reprezinta cele doua apeluri recursive din algoritmul QuickSort, in timp ce ultimul reprezinta procesul de partitionare.k reprezinta numarul de elemente mai mici decat pivotul.

In functie de cum este construit vectorul si de strategia de partitionare aleasa, exista 3 cazuri de complexitate:

Worst case:

Cel mai rau caz se intampla atunci cand procesul de partitionare alege pe post de pivot cel mai mare sau cel mai mic element din pivot. Luand in considerare cazul propus in solutia temei, si anume ca pivotul va fi mereu ultimul element din vector, cel mai rau caz ce intampla atunci cand vectorul nostru este sortat crescator sau descrescator. Recurenta ce reprezinta cel mai rau caz este urmatoarea:

$$\mathrm{T(n)} = \mathrm{T(n\text{-}1)} + \theta(n)$$

In cel mai rau caz, QuickSort are o complexitate de $\theta(n^2)$.

Best case:

Cel mai bun caz se intampla cand procesul de partionare alege pivotul drept mijlocul vectorului, si este reprezentat prin urmatoarea recurenta:

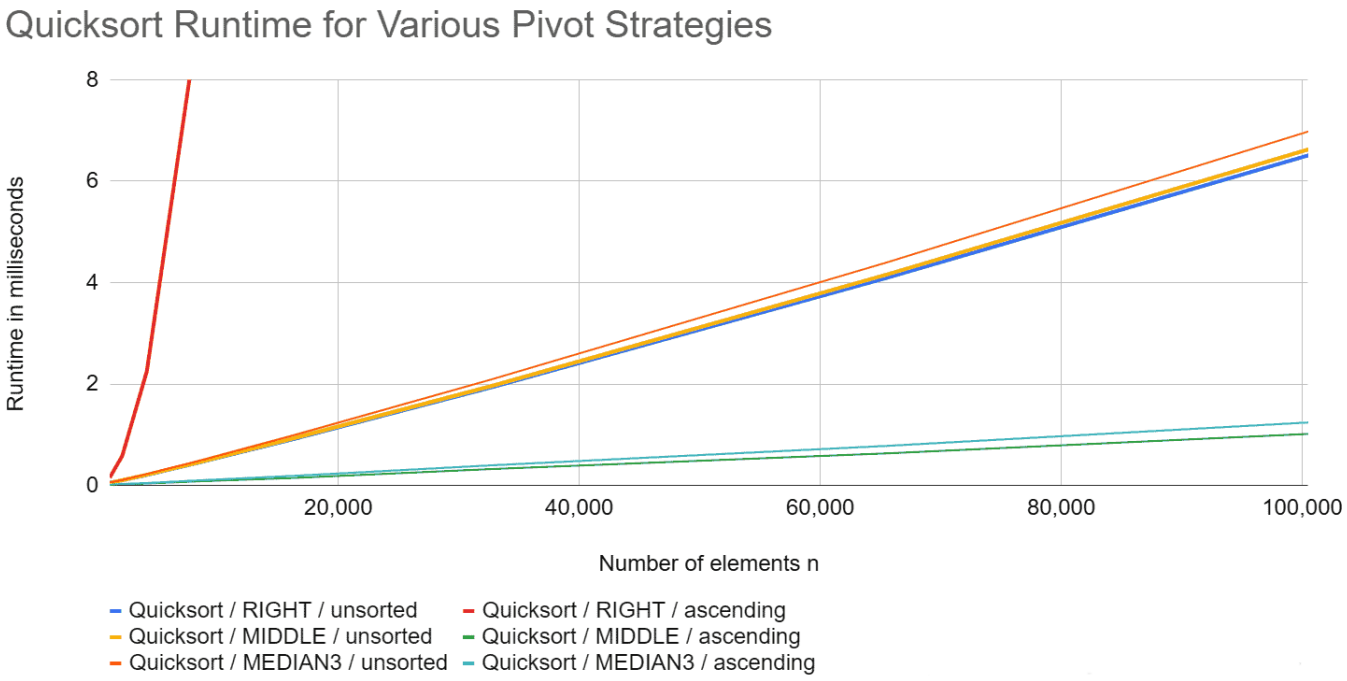
$$\mathrm{T(n)} = 2\mathrm{T(n/2)} + \theta(n)$$

Urmatoarea solutie are o complexitate de θ(nLogn), ce poate fi determinata prin recurenta de mai sus, aplicand cazul 2 al Teoremei Master.

1.5 Eficienta algoritmului

Chiar daca in cel mai rau caz are o complexitate de $\theta(n^2)$, in viata reala QuickSort este destul de eficient, deoarece are un avantaj fata de alti algoritmi de sortare, si anume pivotul. In functie de cum ne alegem pivotul putem evita de cele mai multe ori Worst Case, desi pentru date extrem de mari se prefera alti algoritmi de sortare.

O alta problema legata de QuickSort este ca in cele mai simple cazuri, chiar si atunci cand alegem un pivot pentru obtinerea cazului cel mai favorabil, pot sa apara probleme de stabilitate. QuickSort nu face diferenta intre elemente ce au chei egale, si le inverseaza si pe ele, deci pentru un vector cu elemente egale, ele nu vor aparea in aceeasi ordine dupa sortare. Pentru a oferi stabilitate algritmului ar trebui sa fie implementat intr-un fel ce nu permite apelarea functiei de swap pentru elemente egale.



1.6 Exemplu - QuickSort

```
v[] = {10, 80, 30, 90, 40, 50, 70}
Index:  0   1   2   3   4   5   6

start = 0, end =  6, pivot = v[end] = 70
Initializam indexul celui mai mic element la: i = -1

Iteram cu j de la start la end - 1
j = 0 : v[j] <= pivot ==> i++ si swap(v[i], v[j])
i = 0
v[] = {10, 80, 30, 90, 40, 50, 70} // i == j, nicio schimbare

j = 1 : v[j] > pivot, // nicio schimbare

j = 2 : v[j] <= pivot ==> i++ si swap(v[i], v[j])
i = 1
v[] = {10, 30, 80, 90, 40, 50, 70} // Facem swap intre 80 si 30

j = 3 : v[j] > pivot // nicio schimbare

j = 4 : v[j] <= pivot ==> i++ si swap(v[i], v[j])
i = 2
v[] = {10, 30, 40, 90, 80, 50, 70} // Facem swap intre 80 si 40

j = 5 : v[j] <= pivot ==> i++ si swap(v[i], v[j])
i = 3
v[] = {10, 30, 40, 50, 80, 90, 70} // Facem swap intre 90 si 50


In final, vom pune pivotul la pozitia corecta, facand swap intre
v[i+1] si v[end]
v[] = {10, 30, 40, 50, 70, 90, 80} // Facem swap intre 80 si 70

Rezultat final ==>

v[] = {10, 30, 40, 50, 70, 90, 80}
```

2 Tehnica Greedy

2.1 Enuntul problemei

Se dă un grup de k oameni care vor să cumpere împreună n flori. Fiecare floare are un preț de bază, însă prețul cu care este cumpărată variază în funcție de numărul de flori cumpărate anterior de persoana respectivă. De exemplu dacă George a cumparat 3 flori (diferite) și vrea să cumpere o floare cu prețul 2, el va plăti $(3+1) \times 2 = 8$. Practic el va plăti un preț proporțional cu numărul de flori cumpărate până atunci tot de el.

Cerinta:

Se cere pentru un număr k de oameni și n flori să se determine care este costul minim cu care grupul poate să achiziționeze toate cele n flori o singură dată.

Observatie:

Un tip de floare se cumpără o singură dată. O persoană poate cumpăra mai multe tipuri de flori. În final în grup va exista un singur exemplar din fiecare tip de floare.

Formal avem k număr de oameni, n număr de flori, c[i] = pretul florii de tip i, costul de cumpărare i va fi $(x + 1) \times c[i]$, unde x este numărul de flori cumpărate anterior de persoana respectivă.

2.2 Descrierea solutiei problemei

Se observă că prețul efectiv de cumpărare va fi mai mare cu cât cumpărăm acea floare mai tarziu. Dacă considerăm cazul în care avem o singură persoană în grup observăm că are sens să cumpărăm obiectele în ordine descrescătoare (deoarece vrem să minimizăm costul fiecărui tip de floare, acesta crește cu cât cumpărăm floarea mai tarziu).

De aici, gândindu-ne la versiunea cu k persoane, observăm că ar fi mai ieftin dacă am repartiza următoarea cea mai scumpa floare la alt individ. Deci împărțim florile sortate descrescător după pret în grupuri de câte k, fiecare individ luând o floare din acest grup și ne asigurăm că prețul va creste doar in funcție de numărul de grupuri anterioare.

2.3 Pseudocodul algoritmului

```
struct greater_comparator
{
    template<class T>
    bool operator()(T const &a, T const &b) const { return a > b; }
};
```

```
int minimum_cost(int k, int costs[]) {

    // Sortam vectorul de costuri in ordine descrescatoare
    sort(costs.begin(), costs.end(), greater_comparator());

    // Numarul de flori cumparate de fiecare individ din grup
    int x = 0;
    int cost_total = 0;

    // Parcurgem fiecare cost de floare si il "asignam" unui individ din grup
    // pretul acesteia fiind proportional cu numarul de achizitii facut pana acum
    // de acesta (x)
    for (int i = 0; i < costs.size(); i++) {
        int customer_idx = i % k;
        cost_total += (x + 1) * costs[i];
        // Atunci cand ultimul individ a cumparat o floare din grupul curent
        // incrementam numarul de flori achizitionate de fiecare persoana
        if (customer_idx == k - 1) {
            x += 1;
        }
    }
    return cost_total;
}
```

O varianta mai putin eficienta spatial ar fi fost sa retinem pentru fiecare individ din grup numarul de flori cumparate intr-un hashmap.

2.4 Complexitatea algoritmului

Solutia pentru aceasta problema are o complexitate temporala de $T(n) = O(n * log(n))$, deoarece sortarea are o complexitate de $O(n*log(n))$, plus inca o parcurgere ce are o complexitate de O(n).

Complexitatea spatiala depinde de tipul de sortare ales in implementare, fara partea de sortare spatiul este constant.

2.5 Eficienta algoritmului si Optimul Global

Din punct de vedere al eficientei, algoritmul este stabil, cu mentiunea ca este importanta si stabilitatea algoritmului de sortare. De exemplu, pentru un algoritm de sortare instabil, precum QuickSort, unde algoritmul face swap intre elemente cu cheile egale, ar putea sa apara probleme pentru vectori unde doua flori diferite au acelasi pret.

Algoritmul specificat respecta proprietatea de alegere de tip Greedy, pentru ca inca de la inceputul lui, vectorul de costuri pentru fiecare floare este sortat descrescator. De asemenea, proprietatea de substructura optimala este indeplinita: la fel cum pentru o singura persoana putem sorta descrescator vectorul de preturi, pentru k persoane putem sorta vectorul de preturi descrescator in grupuri de cate k.

2.6 Exemplu - Problema florarului

```
n = 3, k = 3, c = {2, 5, 6}

Sortam vectorul descrescator ==>

n = 3, k = 3, c = {6, 5, 2}

x = 0
cost_total = 0

_____________________________________

i = 0 :

customer_idx = i % k = 0 % 3 = 0
cost_total = cost_total + c[0] = 6

customer_idx != k - 1 (== 2)

_____________________________________

i = 1 :

customer_idx = 1 % k = 1 % 3 = 1
cost_total = 6 + 5 = 11

customer_idx != k - 1 (== 2)

_____________________________________

i = 2 :

customer_idx = 2 % k = 2 % 3 = 2
cost_total = 11 + 2 = 13

customer_idx == k - 1 (== 2)

==> x = x + 1 = 1
```

3 Tehnica Programarii Dinamice

3.1 Enuntul problemei

Fie n prieteni. Un prieten poate sa raman singur sau poate fi pus intr-o pereche cu un alt prieten. Sa se determine toate variantele in care prietenii pot sa ramana singuri sau pot fi grupati cu un alt prieten.

3.2 Descrierea solutiei problemei

O solutie eficienta a acestei probleme presupune folosirea programarii dinamice, pentru a gestiona exact toate combinatiile in care un singur prieten poate fi distribuit cu restul prietenilor.

Pornind de la urmatoarea formula de recurenta: f(n) = f(n - 1) + (n - 1) * f(n - 2), unde f(n) reprezinta modalitatile in care n persoane pot ramane singure sau in pereche cu altcineva, putem determina toate combinatiile posibile.

De exemplu, pentru 3 persoane, exista 4 cazuri unice in care ele se pot grupa:

```
{1}, {2}, {3} : toti sunt singuri
{1}, {2, 3} : 2 si 3 formeaza o pereche, dar 1 este singur
{1, 2}, {3} : 1 si 2 formeaza o pereche, dar 3 este singur
{1, 3}, {2} : 1 si 3 formeaza o pereche, dar 2 este singur
```

3.3 Pseudocodul algoritmului

```
prieteni(n)
{
    dp[n + 1]; //initializam vectorul dinamic

    //Umplem vectorul dinamic dupa formula de recurenta
    // descrisa mai sus
    for (i = 0; i <= n; i++) {
        if (i <= 2)
            dp[i] = i;
        else
            dp[i] = dp[i - 1] + (i - 1) * dp[i - 2];
    }

    return dp[n];
}
```

3.4 Complexitatea algoritmului

Din moment ce vom itera intreg vectorul dinamic, algoritmul va avea o complexitate temporala si spatiala de O(n).

3.5 Determinarea relatiei de recurenta

Presupunem ca f(n) = modalitatile in care n oameni pot ramane singuri sau grupati in perechi cu alti oameni.

Pentru a n-a persoana vor exista doua variante: ramane singura deci vom calcula toate posibilitatile de grupare pentru n - 1 persoane, sau va forma o pereche cu una dintre cele n - 1 persoane caz in care numarul de combinatii va fi (n - 1) * f(n - 2)

Recurenta finala va fi:

$$f(n) = f(n - 1) + (n - 1) * f(n - 2)$$

3.6 Exemplu - Problema Prieteni

```
n = 3 //Numarul de prieteni

//Initializam vectorul dinamic
dp[n + 1]

i = 0 :

i <= 2 ==> dp[i] = dp[0] = 0;

----------------------------

i = 1 :

i <= 2 ==> dp[i] = dp[1] = 1;

----------------------------

i = 2 :

i <= 2 ==> dp[i] = dp[2] = 2;

----------------------------

i = 3 :

i > 2 ==>

dp[i] = dp[i - 1] + (i - 1) * dp[i - 2];

----------------------------

==>

dp[3] = dp[2] + 2 * dp[1] = 2 + 2 * 1 = 4;

==> Pritenii se pot grupa in 4 posibilitati (incluzand cazul in care raman singuri)

----------------------------
```

4 Tehnica Backtracking

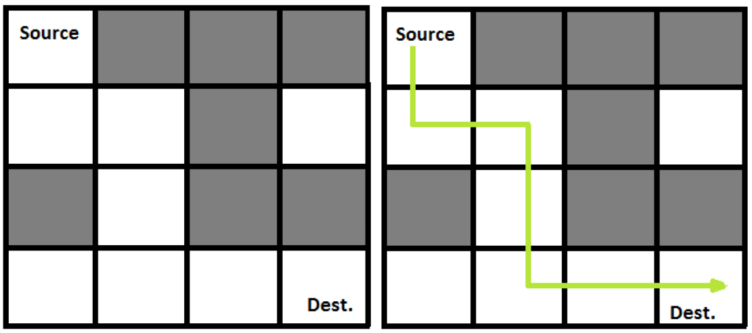
4.1 Enuntul problemei

Fie un labirint reprezentat sub forma unei matrice de NxN, unde N este dimensiunea matricei. Zona de intrare in labirint este reprezentata de pozitia din matrice de la indexul [0][0], iar zona de iesire, prin index ul [N - 1][N - 1]. Un soarece intra in labirint, iar acesta se poate deplasa doar in fata (in dreapta in matrice) sau in spate (in jos in matrice). Sa se determine o cale pe care o poate lua soarece pentru a ajunge de la zona de intrare, la zona de iesire.

Observatie:

In matricea labirintului, o valoare de 0 reprezinta ca pe acel bloc avem o cale care nu duce nicaieri, iar o valoare de 1 reprezinta un bloc care duce spre zona de iesire.

Exemplu:



(a) Matricea labirintului (b) O cale posibila spre iesire

Matricea labirintului de mai sus va fi reprezentata in cod astfel:

```
{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}
```

4.2 Descrierea solutiei problemei

Pentru rezolvarea acestei probleme vom folosi tehnica de Backtracking pentru a genera recursiv, la fiecare iteratie de pe stiva calea cea buna pana la zona de iesire. Daca la pasul curent calea nu este valida, ne intoarcem cu un pas si reanalizam toate caile posibile pe care soarecele le poate urma. Mai exact, algoritmul va functiona dupa urmatorii pasi:

1. Ne definim o matrice de solutii, initial cu elementele pe 0.
2. Ne definim o functie recrusiva ce primeste ca parametrii matrcicea de solutii, matricea de output si pozitia [i][j] din matrice a soarecelui
3. Daca pozitia gasita nu se afla in matrices au nu este valida return.
4. Marcam cu 1 pozitia [i][j] din matricea de output si verificam daca este pozitia zonei de iesire; daca este printam matrciea de output si iesim din functie.
5. Apelam recursive pentru pozitiile [i + 1][j] si [i][j + 1].
6. Marcam cu 0 pozitia [i][j] din matricea de output.

4.3 Pseudocodul algoritmului

```
bool solveMazeUtil(
    int maze[N][N], int x,
    int y, int sol[N][N]);


void printSolution(int sol[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            print(sol[i][j]);
    }
}

bool isSafe(int maze[N][N], int x, int y)
{
    // daca (x, y sunt in afara labirintului) returnam false
    if (
        x >= 0 && x < N && y >= 0
        && y < N && maze[x][y] == 1)
        return true;

    return false;
}

bool solveMaze(int maze[N][N])
{
    int sol[N][N] = { { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 } };

    if (solveMazeUtil(
            maze, 0, 0, sol)
        == false) {
        print("Solution doesn't exist");
        return false;
    }

    printSolution(sol);
    return true;
}
```

```
bool solveMazeUtil(
   int maze[N][N], int x,
   int y, int sol[N][N])
{
   // daca (x, y este destinatia) returnam true
   if (
       x == N - 1 && y == N - 1
       && maze[x][y] == 1) {
       sol[x][y] = 1;
       return true;
   }

   // Verificam daca maze[x][y] este valid
   if (isSafe(maze, x, y) == true) {
         if (sol[x][y] == 1)
             return false;

       sol[x][y] = 1;

       if (solveMazeUtil(
               maze, x + 1, y, sol)
           == true)
           return true;

       if (solveMazeUtil(
               maze, x, y + 1, sol)
           == true)
           return true;

       if (solveMazeUtil(
               maze, x - 1, y, sol)
           == true)
           return true;

       if (solveMazeUtil(
               maze, x, y - 1, sol)
           == true)
           return true;

       sol[x][y] = 0;
       return false;
   }

   return false;
}
```

4.4 Complexitatea algoritmului

Algoritmul ce rezolva problema din enunt are o complexitate temporala ce poate ajunge in worst case in $O(2^{(n^2)})$, deoarece recursivitatea pe matrcie poate rula de maxim $2^{(n^2)} ori$.

Din punct de vedere al spatiului, algoritmul are o complexitate de $O(n^2)$,deoarece ne trebuie si o matrice de output, deci ne trebuie un spatiu aditional de n * n.

Algoritmul ce rezolva problema din enunt are o complexitate temporala ce poate ajunge in worst case in $O(2^{(n^2)})$, deoarece recursivitatea pe matrcie poate rula de maxim $2^{(n^2)} ori$.

4.5 Exemplu - Problema Labirintului

```
n = 4;

maze[n][n] = { 1, 0, 0, 0 },
             { 1, 1, 0, 1 },
             { 0, 1, 0, 0 },
             { 1, 1, 1, 1 }


sol[n][n] = { 0, 0, 0, 0 },
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 }


Pasul 1:

 0  0  0  0
 0  0  0  0
 0  0  0  0
 0  0  0  0

Pasul 2:

 1  0  0  0
 0  0  0  0
 0  0  0  0
 0  0  0  0

Pasul 3:

 1  0  0  0
 1  0  0  0
 0  0  0  0
 0  0  0  0

Pasul 4: // In jos nu este valid ==> mergem in dreapta

 1  0  0  0
 1  0  0  0
 X  0  0  0
 0  0  0  0
```

```
Pasul 5:

 1  0  0  0
 1  1  0  0
 0  0  0  0
 0  0  0  0

Pasul 6:

 1  0  0  0
 1  1  0  0
 0  1  0  0
 0  0  0  0

Pasul 7:

 1  0  0  0
 1  1  0  0
 0  1  0  0
 0  1  0  0

Pasul 8: // In jos nu mai putem merge (am depasi matrciea)
          // ==> mergem in dreapta

 1  0  0  0
 1  1  0  0
 0  1  0  0
 0  1  0  0

Pasul 9:

 1  0  0  0
 1  1  0  0
 0  1  0  0
 0  1  1  0

Pasul 10:

 1  0  0  0
 1  1  0  0
 0  1  0  0
 0  1  1  1

==> Am obtinut o cale posibila.
```

5 Analiza Comparativa

5.1 Aplicativitatea tehnicilor pentru tipuri probleme

Divide et Impera:

Tehnica Divide et Impera este folosita extrem de des pentru implementarea algoritmilor in care este eficient sa impartim problema curenta in mai multe subprobleme. De exemplu algoritmii de sortare beneficiaza extrem de mult pe baza Divide et Impera. Exista totusi o limitare. Pe principiul impartirii unei probleme, mai exact, intr-un mod recursiv, putem ajunge la concluzia ca orice algoritm recursiv este de tipul Divide et Impera, ceea ce nu este corect. Este considerat un algoritm divide daca genereaza doua sau mai multe subprobleme.

Greedy:

Tehnica Greedy presupune impartirea problemei sau al unui set de resurse, in functie de cea mai mare valoare a acelei resurse aflata la cea mai apropiata distribuitate (Valoare maxima in cel mai scurt timp posibil). Algoritmii lacomi sunt algoritmi instinctivi simpli folosiți pentru probleme de optimizare (fie maximizate, fie minimizate). Acest algoritm face cea mai bună alegere la fiecare pas și încearcă să găsească modalitatea optimă de a rezolva întreaga problemă. Este important de mentionat ca nu intotdeauna o solutie pe care aplicam logica greedy este si optima, in schimb putem obtine o solutie ce aproximata de apropie de un optim global.

Programare dinamica:

Programarea dinamică este o tehnică pentru rezolvarea problemelor cu subprobleme suprapuse. Fiecare subproblemă este rezolvată o singură dată și rezultatul fiecărei subprobleme este stocat într-un tabel (în general implementat ca o matrice sau un hashtable) pentru referințe viitoare. Aceste subsoluții pot fi utilizate pentru a obține soluția originală și tehnica de stocare a soluțiilor de subproblemă este cunoscută sub numele de memorizare. Tehnica de memorizare sugereaza astfel ca acest tip de algoritm este folosit pentru a gasi solutii optime la problema noastra.

Backtracking:

În general, orice problemă de satisfacție a constrângerilor clare și bine definite, asupra oricărei soluții obiective, ce creează treptat solutia candidat și o abandonează la fel de repede atunci cand determină că respectivul candidat nu poate fi completat cu o soluție, poate fi rezolvată prin Backtracking. Cu toate acestea, majoritatea problemelor discutate pot fi rezolvate folosind alți algoritmi cunoscuți, cum ar fi programare dinamică sau algoritmi greedy într-o complexitate logaritmica, liniara, sau liniar-logaritmica, în ordinea dimensiunii intrării și, prin urmare, depășesc algoritmul de urmărire inversă din toate punctele de vedere (deoarece algoritmii de backtracking sunt în general exponențiali atât în timp cât și în spațiu).

5.2 Avantaje si dezavantaje pentru fiecare tehnica

Divide et Impera

Avantaje:

1. Rezolvarea problemelor dificile.
2. Eficienta.
3. Paralelism
4. Gestionarea mai eficienta a memoriei.
5. Acuratete mai buna pentru floating-point numbers.

Dezavantaje:

1. Recursivitatea poate deveni inceata, negand pasii facuti spre o eficienta mai buna.
2. Pentru numere foarte mari, e preferata o solutie iterativa.

Greedy

Avantaje:

1. Usor de ales cea mai buna solutie optima.
2. Mai usor de analizat timpii de rulare.

Dezavantaje:

1. Greu de demonstrat.
2. Nu este recomandata pentru probleme unde solutia este necesara pentru fiecare subproblema.

Programare dinamica

Avantaje:

1. Este potrivit pentru procesele de decizie în mai multe etape sau în mai multe puncte sau procesele secvențiale.
2. Este potrivit pentru probleme liniare sau neliniare, variabile discrete sau continue și probleme deterministe.

Dezavantaje:

1. Fiecare problema (recurenta) trebuie rezolvata intr-un mod diferit, nu exista o "reteta" pentru aplicarea programarii dinamice.
2. Consuma multa memorie.

Backtracking

Avantaje:

1. Putem determina toate solutiile existente pentru o problema.
2. Usor de implementat.

Dezavantaje:

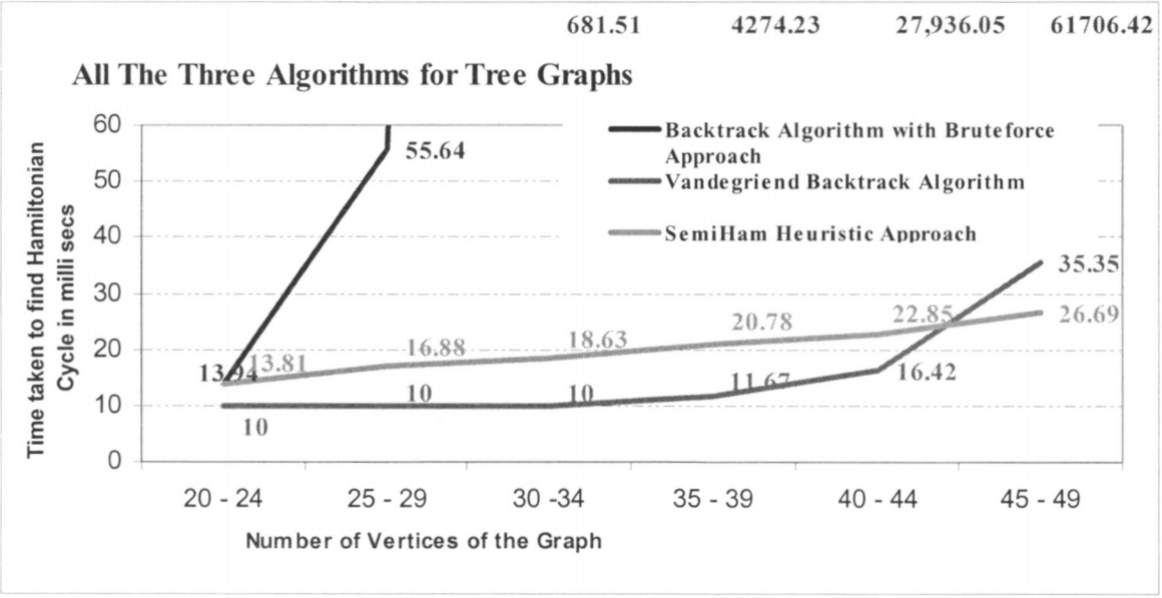
1. Exista solutii mai eficiente folosind PD sau Greedy.
2. Complexitate spatiala foarte mare.
3. Greu de optimizat.
4. Poate rula in timp polinomial.

5.3 Probleme ce pot fi rezolvate prin mai multe tehnici

Cand vine vorba de rezolvarea unei probleme, o intrebare ce apare imediat in procesul de implementare va fi: Ce algoritm este cel mai eficient pentru rezolvarea problemei mele? Cele 4 tehnici prezentate de-a lungului temei, precum si problemele prezentate sunt un pas de inceput foarte bun pentru alegerea solutiei optime, dar exista o problema, si anume problemele ce se pot rezolva la fel de usor folosind mai multe tehnici.

O astfel de problema, este un algoritm clasic de parcurgere a grafurilor, mai exact algoritmul ciclurilor Hamiltoniene. Aceasta problema poate fi rezolvata prin backtracking, avand o complexitate in worst case prin DFS de O(n!). Aceasta este solutia optima pentru aceasta problema, insta exista mai multe variante de rezolvare a acestei probleme. Putem folosi de exemplu Programarea Dinamica, dar cu un rezultat mai ineficient, cu o complexitate de $O(n^2 * 2^n)$. În această metodă, se determină, pentru fiecare set S de vârfuri și fiecare vârf v în S, dacă există o cale care acoperă exact vârfurile din S și se termină la v.

De asemenea mai exista si un algoritm euristic mai eficient decat Backtracking sau orice alta solutie oferita de tehnicile din tema ce ofera acelasi rezultat, dar de 5 ori mai repede.



References

[1] GeeksforGeeks, QuickSort,

Disponibil aici:
https://www.geeksforgeeks.org/quick-sort/

[2] Quora, Stabilitate QuickSort,

Disponibil aici:
https://www.quora.com/Is-quick-sort-a-stable-sorting-algorithm

[3] HappyCoders.eu, Eficienta QuickSort,

Disponibil aici:
https://www.happycoders.eu/algorithms/quicksort/

[4] ocw.cs.pub.ro, Probleme Greedy,

Disponibil aici:
https://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-02

[5] GeeksforGeeks, Friends pairing problem,

Disponibil aici:
https://www.geeksforgeeks.org/friends-pairing-problem/

[6] GeeksforGeeks, Rat in a Maze,

Disponibil aici:
https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/

[7] Wikipedia, Divide et Impera,

Disponibil aici:
https://en.wikipedia.org/wiki/Divide-and-conquer$_algorithm$

[8] guru99, Greedy,

Disponibil aici:
https://www.guru99.com/greedy-algorithm.html

[9] StackOverflow, Difference between Divide and Conquer Algo and Dynamic Programming,

Disponibil aici:
$https://stackoverflow.com/questions/13538459/difference-between-divide-and-conquer-algo-and-dynamic-programming$

[10] Kodnest, Advantages and Disadvantages of Divide and Conquer,

Disponibil aici:
$https://www.kodnest.com/free-online-courses/algorithm-2/lessons/advantages-and-disadvantages-of-divide-and-conquer/$

[11] medium.com, When to use Greedy Algorithms in Problem Solving,

Disponibil aici:
shorturl.at/uxN78

[12] Quora, What are the advantages and disadvantages of dynamic programming?,

Disponibil aici:
https://www.quora.com/What-are-the-advantages-and-disadvantages-of-dynamic-programming

[13] Quora, What are the advantages and disadvantages of a backtracking algorithm?,

Disponibil aici:
https://www.quora.com/What-are-the-advantages-and-disadvantages-of-a-backtracking-algo

[14] hackerearth, Hamiltonian Path,

Disponibil aici:
https://www.hackerearth.com/practice/algorithms/graphs/hamiltonian-path/tutorial/

[15] Wikipedia, Hamiltonian path problem,

Disponibil aici:
https://en.wikipedia.org/wiki/Hamiltonian$_p ath_p roblem$

[16] The Faculty of the Department of Computer Science Western Kentucky University Bowling Green, Kentucky, FINDING HAMILTONIAN CYCLES ,

Disponibil aici:
https://core.ac.uk/download/pdf/43618678.pdf