Rosu Adriana-Stefania
Grupa 324CD
Facultatea de Automatica si Calculatoare
Anul II

Tema 0

Proiectarea Algorimilor

Tehnici de programare

27 aprilie 2021

Cuprins

1 Divide et Impera

1.1 Numărul de zile după care rezervorul va deveni gol

1.1.1 Enuntul problemei

Se da un rezervor cu capacitate C litri care este complet umplut la inceput. Rezervorul este umplut in fiecare zi cu l litri de apă și în caz de revărsare se aruncă apă suplimentară. In a i-a zi, se iau i litri de apă pentru băut. Trebuie să aflăm ziua în care rezervorul va deveni gol prima dată.

1.1.2 Descrierea solutiei problemei

Initial, m-am gândit să adaug l litri în rezervor și să scot i litri, unde i reprezintă numărul zilei în care ne aflăm. După aceasta, verificăm pur și simplu dacă rezervorul este gol.
După o privire mai atentă, mi-am dat seama că nu este o soluție atât de optimizată. Am aflat că în primele l + 1 zile (unde l este numărul de litri pe care îi adăugăm în fiecare zi) rezervorul va fi plin, deoarece numărul de litri pe care îl scoatem este mai mic decât ceea ce adăugăm. După ultima zi in care rezervorul este plin, numărul de litri va scădea în mod constant cu unul mai mult decât în ziua precedentă. In acest caz observam faptul ca in ziua numarul (l + 1 + i), in rezevor se vor afla, inainte de a scoate apa de baut, un numar de $C - \dfrac{(i)(i+1)}{2}$ litri (i reprezinta numarul zilei de dupa cea in care rezervorul nu mai este plin confrom regulii de mai sus si $\dfrac{(i)(i+1)}{2}$ reprezinta suma zilelor precedente).Pentru a rezolva problema noastră, trebuie să găsim ziua urmatoare celor l + 1 în care, chiar și după umplerea rezervorului cu l litri, avem în rezervor mai putina apă decat l litri. Dupa aceasta, vom sti ca ziua anterioara celei mentionate este ziua in care rezervorul va fi gol. Deci scopul nostru este sa gasim K minim pentru care $C - \dfrac{K(K+1)}{2} \leq$ l.
Putem găsi soluția ecuației folosind căutarea binară și (l + K) va fi răspunsul.

1.1.3 Prezentarea algoritmului de rezolvare a problemei

Algorithm 1 Pseudocod

```
    https://www.overleaf.com/project/6070c3c8f977ba0dcf1473d6  if C ≤ l then
      return C
  end if
  low ← 0
  high ← inf
  while low < high do
      middle ← (low + high)/2
      if middle * (middle + 1)/2 ≥ C − 1 then
          high ← middle
      else
          low ← middle + 1
      end if
  end while
  return low + l
  =0
```

1.1.4 Complexitatea algoritmului

Complexitatea timpului total al soluției va fi O(log C).

1.1.5 Analiza succinta asupra eficientei algoritmului propus

Prima soluție pe care am prezentat-o este cea mai ineficientă. Există o altă soluție în afară de cea propusă de mine pe baza unei formule matematice directe.

$$minDays = L + ceil(\frac{\sqrt{1 - 8(C - L)} - 1}{2}) \tag{1}$$

1.1.6 Exemplificarea aplicarii algoritmului propus

- - Date de intrare :
  - C = 10
  - l = 3

  - Date de iesire : 7
- - Rezolvare :
  - Apa din rezervor la inceputul primei zile = 10 si la sfarsitul primei zilei = (10 - 1) = 9
  - Apa din rezervor la inceputul zilei 2 = 9 + 3 = 12 dar capacitatea rezervorului este 10 deci, apa = 10 si la sfarsitul zilei 2 = (10 - 2) = 8
  - Apa din rezervor la inceputul zilei 3 = 8 + 3 = 11 dar capacitatea rezervorului este 10 deci, apa = 10 si la sfarsitul zilei 3 = (10 - 3) = 7
  - Apa din rezervor la inceputul zilei 4 = 7 + 3 = 10 si la sfarsitul zilei 4 = (10 - 4) = 6
  - Apa din rezervor la inceputul zilei 5 = 6 + 3 = 9 si la sfarsitul zilei 5 = (9 - 5) = 4
  - Apa din rezervor la inceputul zilei 6 = 4 + 3 = 7 si la sfarsitul zilei 6 = (7 - 6) = 1
  - Apa din rezervor la inceputul zilei 7 = 1 + 3 = 4 si la sfarsitul zilei 7 = (4 - 7) ¡ 0
- Rezultatul final va fi 7.

2 Greedy

2.1 Colorarea grafului

2.1.1 Enuntul problemei

Se dau m culori. Sa se gaseasca o modalitate de a colora vârfurile unui graf astfel încât să nu fie colorate două vârfuri adiacente folosind aceeași culoare.

2.1.2 Descrierea solutiei problemei

Se va colora primul varf cu prima culoare. Pentru varfurile ramase, vom colora fiecare varf in parte cu cea mai mica numarotata culoare ce nu a fost folosita anterior pentru a colora varfurile adiacente. Daca toate culorile precedente au fost folosite pentru a colora varfurile adiacente, atunci ii vom atribui o noua culoare.

2.1.3 Prezentarea algoritmului de rezolvare a problemei

Algorithm 2 Pseudocod

```
for i = 1..V do
  result[i] ← −1
end for
result[0] ← 0
for i = 1..m do
  available[i] ← true
end for
for i = 1..(V − 1) do
  for j = 1..n do
    if result[j] ≠ −1 then
      available[result[i]] ← false
    end if
  end for
  for k = 1..V do
    if available[i] then
      break
    end if
  end for
  result[i] ← k
end for
for i = 1..m do
  available[i] ← true
end for
=0
```

2.1.4 Complexitatea algoritmului

Complexitatea timpului total al soluției va fi O(V^2 + E), in cel mai rau caz.
Algoritmul de mai sus nu folosește întotdeauna un număr minim de culori. De asemenea, numărul de culori utilizate depinde câteodata de ordinea în care sunt procesate vârfurile. Problema ar putea fi optimizata prin folosirea unui algoritm bazat pe BFS.

2.1.5 Analiza succinta asupra posibilităti de obtinere a optimului global

Algoritmul face alegeri care pot depinde de alegerile facute anterior(de culorile alese anterior), dar nu si de viitoarele alegeri sau de toate solutiile subproblemlor. Acesta face iterativ o alegere greedy dupa si reduce de fiecare data problema la una mai mic. Astfel, este indeplinita proprietatea de alegere de tip Greedy.
Proprietatea de substructură optimală se refera la faptul ca daca solutia problemei este una optima, atunci solutia unei subprobleme va fi si ea optima. Acest lucru se intample deoarce la fiecare pas se alege cea mai optima metoda de a colora nodurile.

2.1.6 Exemplificarea aplicarii algoritmului propus

- - Date de intrare :
  - m = 3
  - V = 5
  - E = 6

  - Date de iesire : 7

- - Rezolvare :
  - varful 0 - culoarea 0
  - varful 1 - culoarea 1
  - varful 2 - culoarea 2
  - varful 3 - culoarea 0
  - varful 4 - culoarea 1

3 Programare dinamica

3.1 Numarul minim de sarituri pentru a ajunge la final

3.1.1 Enuntul problemei

Dandu-se un vector de numere intregi in care fiecare element reprezinta numarul maxim de pasi ce pot fi facuti inainte fata de acel element. Problema va gasi cel mai mic numar de sarituri necesare pentru a ajunge la finalul vectorului, incepand de la primul element. Daca un element este 0, atunci nu se poate trece prin el. Daca nu se poate ajunge la final problema va returna -1.

3.1.2 Descrierea solutiei problemei

Solutia ce foloseste programrea dinamica presupune crearea unui nou vector ce va retine pe pozitia i, numarul minim de sarituri necesar pentru a ajunge de pe prima pozitie pana pe pozitia i in vectorul dat initial. Astfel, rezultatul problemei se va afla pe ultima pozitia a vectorului nou creat.

3.1.3 Prezentarea algoritmului de rezolvare a problemei

Algorithm 3 Pseudocod

if $n = 0$ **then**
  return Nu se poate ajunge la final.
end if
if $arr[0] = 0$ **then**
  return Nu se poate ajunge la final.
end if
$jumps[0] \leftarrow 0$
for $i = 1..(n-1)$ **do**
  $jumps[0] \leftarrow \infty$
  for $j = 0..(i-1)$ **do**
    if $i \leq j + arr[j]$ & $jumps[j] \neq \infty$ **then**
      $jumps[i] \leftarrow min(jumps[i], jumps[j] + 1)$
      $break$
    end if
  end for
end for
return jumps[n - 1];
=0

3.1.4 Complexitatea algoritmului

Complexitatea timpului total al soluției va fi O(n^2). In ceea ce priveste memoria, va fi o complexitate de O(n) necesara stocarii vectorului DP.
Folosirea programarii dinamice reprezinta cea mai eficienta solutie de rezolvare a problemei, avand o complexitate semnificativ mai mica decat cea a unei abordari recursive de O(n^n).

3.1.5 Explicarea modului în care a fost obtinută relatia de recurenta

- Cazul de baza: Daca vectorul nu are elemente sau primul element este 0 (nu se poate sari de la el mai departe), atunci nu se poate ajunge la destinatie.
  - n = 0 sau arr[0] = 0 : nu se poate sari pana la sfarsitul vectorului.
- Cazul general: La pasul i, se alege cea mai buna solutie pentru a ajunge de la primul element la elementul i. Acest lucru se realizarea prin alegerea minimului dintre numarul curent de sarituri pana la elementul i si numarul de pasi pana la elementele anterioare + 1.
  - jumps[i] = Math.min(jumps[i], jumps[j] + 1), unde j = 0,..,(i - 1), i ¡= j + arr[j] si exista "drum" pana la elementul j.

3.1.6 Exemplificarea aplicarii algoritmului propus

- - Date de intrare :
  - arr[] = {1, 4, 5, 2, 1, 4, 5, 6, 3, 8}

  - Date de iesire : 3
- - Rezolvare :
  - De la primul element se poate sari doar la urmatorul. Astfel, vom avea 1-4.
    De la al doilea element de poate sari la 5 , 2 , 1 sau 4. Deoarce de la elementul 4 se va ajunge cel mai repede la finalul vectorului, vom avea 1-4-4.
    De la elementul 4, se poate ajunge la finalul vectorului deoarece mai sunt necesare doar 4 sarituri. Astfel, sariturile finale, in numar de 3, vor fi 1-4-4-8.

4 Backtracking

4.1 Problema soricelului in labirint

4.1.1 Enuntul problemei

Se da un labirint sub forma unei matrice labirint[N][N], unde labirint[0][0] reprezinta punctul de plecare al unui soricel, iar labirint[N - 1][N -1] reprezinta iesirea din acesta, destinatia in care trebuie sa ajunga soricelul. In labirint exista celule prin care nu se poate trece, acestea fiind marcate cu 0.
Trebuie sa gaseasca toate solutiile de a iesi din labirint, stiind ca soricelul se poate misca doar in doua directii: in sus si in jos.

4.1.2 Descrierea solutiei problemei

Se creeaza o noua matrice reprezentativa solutiilor, solutie[N -1][N -1]ce va fi initializata cu 0. Cu ajutorul unei functii recursive ce va lua labirintul, matricea creata si pozitia soricelului la un anumit pas (i,j), se va verifica initial daca pozitia este valida (daca se afla in matrice). Apoi, solutie[i][j] va primi 1 (astfel se marcheaza drumul soricelului prin labirint), si se va verifica daca s-a ajuns la destinatie. Daca nu, se apela recursiv functia pentru cele doua directii de deplasare (i + 1, j) si (i, j + 1). Daca nu se gaseste o solutie pentru acestea, atunci solutie[i][j] va redeveni egala cu 0 semn ca nu se poate ajunge la un drum daca se trece pe acolo.

4.1.3 Prezentarea algoritmului de rezolvare a problemei

Algorithm 4 solveHelper(int labirint[N - 1][N - 1], int x, int y, int solutie[N - 1][N - 1])

if $x == N-1$ & $y == N-1$ & $labirint[x][y] == 1$ **then**
  $solutie[x][y] \leftarrow 1$
  return
end if
if $isValid(labirint, x, y) ==$ **then**
  if $solutie[x][y] == 1$ **then**
    return false
  end if
  $solutie[x][y] \leftarrow 1$
  if $solveHelper(labirint, x+1, y, solutie) ==$ **then**
    return
  end if
  if $solveHelper(labirint, x-1, y, solutie) ==$ **then**
    return
  end if
  if $solveHelper(labirint, x, y+1, solutie) ==$ **then**
    return
  end if
  if $solveHelper(labirint, x, y-1, solutie) ==$ **then**
    return
  end if
  $solutie[x][y] \leftarrow 0$
  return false
end if
=0

Algorithm 5 solve(int labirint[N - 1][N - 1])

for $i = 0..(N-1)$ **do**
  for $j = 0..(N-1)$ **do**
    $solutie[i][j] \leftarrow 0$
  end for
end for
if $solveHelper(labirint, 0, 0, solutie) ==$ **false then**
  return false
end if
$printSolutie(solutie)$
return
=0

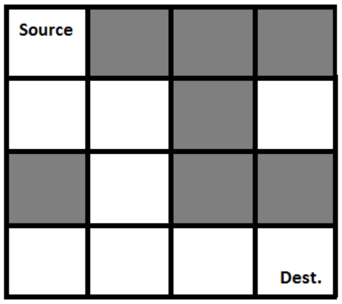
4.1.4 Complexitatea algoritmului

Complexitatea timpului total al soluției va fi O(2^{n^2}).

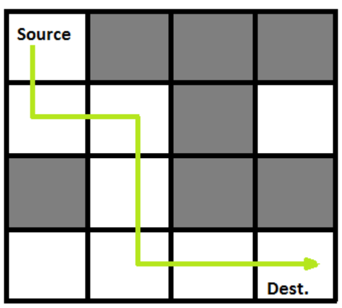
4.1.5 Analiză succinta asupra eficientei algoritmului propus

Solutia folosind backtracking pentru a rezolva problema nu este cea mai optima intrucat foloseste recursivitate cea ce produce o complexitate mare, atat in ce priveste timpul, cat si spatiul (informatia este stocata pe stiva). O solutie mai optima este reprezentate de un algoritm de rezolvare ce foloseste BFS, acesta avand o complexitate lineara.

4.1.6 Exemplificarea aplicarii algoritmului propus



- \- Rezolvare :
  - La primul pas, se poate duce doar in jos.
  - La al doilea pas, se poate duce doar in dreapta.
  - La treilea pas, se poate duce doar in jos.
  - La patrulea pas, se poate duce doar in jos.
  - La cincilea pas, se poate duce in dreapta, insa ulterior va iesi in afara labirintului, asa ca solutia va fi sa se duca in dreapta.
  - La saselea pas, se poate duce doar in dreapta.
  - La saptelea pas, se poate duce doar in dreapta, ajungand la destinatie.



5 Analiza Comparativa

5.1 Tipurile de probleme pentru care fiecare tehnica poate fi aplicata

Divide and Impera se poate aplica la problemele care se pot descompune în subprobleme de aceeaşi natură cu problema principală, in timp ce Greedy se aplică la acele probleme unde datele de intrare sunt organizate sub forma unei mulţimi A şi se cere găsirea unei submulţimi B din A care să îndeplinească anumite condiţii, astfel încât să fie acceptată ca soluţie posibilă. Algoritmii ce folosesc Backtracking sunt aplicati, atunci cand problema cere toate posibilitatile de a se ajunge la o solutie, iar tehnica Programarii Dinamice este folosita pentru rezolvarea problemelor de optimizare, dar se poate folosi și pentru probleme în care nu se cauta un optim, cum ar fi problemele de numărare.

5.2 Avantaje si dezavantaje pentru fiecare tehnica în parte

Tehnica Divide et Impera ne permite să rezolvăm probleme dificile și adesea imposibile. Un avantaj este faptul ca reduce gradul de dificultate, impartind problema în subprobleme ușor de rezolvat. De obicei, rulează mai repede decât ar face alți algoritmi si folosește în mod eficient cache-urile de memorie. Un prim dezavantaj este reprezentat de folosirea recursivitatii lente, iar uneori poate deveni mai complicată decât o abordare iterativă de bază. Un

alt dezavantaj este ca la impartirea in subprobleme, pot aparea mai multe subprobleme la fel ce vor fi rezolvate fiecare.
In ceea ce priveste, tehnica Greedy un avantaj il constituie faptul ca sunt usor de implementat, ei folosind o abordare simpla, rezultand adesea rezultate optimale. Insa, la modul general, Greedy nu găsește soluția optimă și nici măcar nu garantează că se va găsi o soluție, chiar dacă aceasta există, ceea ce reprezinta un dezavantaj.
Un avantaj al Programarii Dinamice este ca, spre deosebire de Greedy, obtine atat optimul global, cat si optimul local, dar si faptul ca nu este recursiv si rezolva o subproblema o singura data (spre deosebire de Divide et Impera). In ceea ce priveste dezavantajele, aceasta tehnica nu prezinta o forma generala, si fiecare problema se rezolva in felul ei dupa deducere unei relatii de recurenta.
Avanjatul Backtracking-ului este ca, spre deosebire de Programarea Dinamica, are o procedura bine definita. Tehnica Backtracking este foarte ineficienta in majoritatea cazurilor, avand o complexitate mare a spațiului, deoarece folosim recursivitatea, astfel încât informațiile despre funcții sunt stocate pe stivă.

6 Referinte

6.1 Divide et Impera

Numărul de zile după care rezervorul va deveni gol

6.2 Greedy

Colorarea grafului

6.3 Programare Dinamica

Numarul minim de sarituri pentru a ajunge la final

6.4 Backtracking

Problema soricelului in labirint