Proiectarea Algoritmilor 2011-2012

Laborator 2-3

Greedy și Programare Dinamică

Cuprins

1 Obiective laborator

- Înțelegerea noțiunilor de bază legate de tehnicile greedy şi programare dinamică;
- Însuşirea abilităților de analiză în vederea conceperii algoritmilor de tip greedy şi programare dinamică pentru rezolvarea diferitelor probleme;
- Însuşirea abilităților de implementare a algoritmilor bazați pe greedy şi programare dinamică.

2 Importanță – aplicații practice

În general tehnicile de tip Greedy sau Programare Dimanică sunt folosite pentru rezolvarea problemelor de optimizare. Acestea pot adresa probleme în sine sau pot fi subprobleme dintr-un algoritm mai mare. De exemplu, algoritmul Dijkstra pentru determinarea drumului minim pe un graf alege la fiecare pas un nod nou urmărind algoritmul greedy.

Exista însă probleme care ne pot induce în eroare. Astfel, există probleme în care urmărind criteriul Greedy nu ajungem la soluția optimă. Este foarte important să identificăm cazurile când se poate aplica Greedy și cazurile când este nevoie de altceva. Alteori această soluție neoptimă este o aproximare suficientă pentru ce avem nevoie. Problemele NP-complete necesita multă putere de calcul pentru a găsi optimul absolut. Pentru a optimiza aceste calcule mulți algoritmi folosesc decizii Greedy și găsesc un optim foarte aproape de cel absolut.

Programarea dinamică are un câmp larg de aplicare, aici amintind genetica (sequence alignment), teoria grafurilor (algoritmul Floyd-Warshall), metode de antrenare a rețelelor neurale (Adaptive Critic training strategy), limbaje formale și automate (algoritmul Cocke-Younger-Kasami, care analizează dacă și în ce fel un șir poate fi generat de o gramatică independent de context), implementarea bazelor de date (algoritmul Selinger pentru optimizarea interogării relaționale) etc.

3 Prezentarea generală a problemei

3.1 Greedy

"greedy" = "lacom". Algoritmii de tip greedy vor să construiască într-un mod cât mai rapid soluția unei probleme. Ei se caracterizează prin luarea unor decizii rapide care duc la găsirea unei soluții potențiale a problemei. Nu întotdeauna asemenea decizii rapide duc la o soluție optimă; astfel ne vom concentra atenția pe identificarea acelor anumite tipuri de probleme pentru care se pot obține soluții optime [8].

Algoritmii greedy se numără printre cei mai direcți algoritmi posibili. Ideea de bază este simplă: având o problema de optimizare, de calcul al unui cost minim sau maxim, se va alege la fiecare pas decizia cea mai favorabilă, fără a evalua global eficiența soluției. În general exista mai multe soluții posibile ale problemei. Dintre acestea se pot selecta doar anumite soluții **optime**, conform unor anumite criterii. Scopul este de a găsi una dintre acestea sau dacă nu este posibil, atunci o soluție cât mai apropiată, conform criteriului optimal impus.

Trebuie înțeles faptul ca rezultatul obținut este optim doar dacă un optim local conduce la un optim global. În cazul în care deciziile de la un pas influențează lista de decizii de la pasul următor, este posibila obținerea unei valori neoptimale. În astfel de cazuri, pentru găsirea unui optim absolut se ajunge la soluții supra-polinomiale. De aceea, dacă se optează pentru o astfel de soluție, algoritmul trebuie însoțit de o demonstrație de corectitudine.

Descrierea formală a unui algoritm greedy este următoarea:

```
function greedy(C)
      // C este mulțimea candidaților
      // în S construim soluția
      S ← Ø
      while not solutie(C) and C≠Ø
            x ← un element din C care minimizează/ minimizează select(x)
            C ← C\{x}
            if fezabil(SU{x}) then S←SU{x}
      return S
```

Este uşor de înţeles acum de ce acest algoritm se numeşte ”greedy”: la fiecare pas se alege cel mai bun candidat de la momentul respectiv, fără a studia alternativele disponibile în moment respectiv şi viabilitatea acestora în timp.

Dacă un candidat este inclus în soluţie, rămâne acolo, fără a putea fi modificat, iar dacă este exclus din soluţie, nu va mai putea fi niciodată selectat drept un potenţial candidat.

3.1.1 Exemplu de problema rezolvabilă cu tehnica Greedy:

Fie un şir de N numere pentru care se cere determinarea unui subşir de numere cu suma maximă. Un subşir al unui şir este format din caractere (nu neapărat consecutive) ale şirului respectiv, în ordinea în care acestea apar în şir.

Pentru numerele 1 -5 6 2 -2 4 răspunsul este 1 6 2 4 (suma 13).

Se observa ca tot ce avem de făcut este sa verificam fiecare număr dacă este pozitiv sau nu. În cazul pozitiv, îl introducem în subşirul soluţie.

3.1.2 Problema cuielor

Fie N scânduri de lemn, descrise ca nişte intervale închise cu capete reale. Găsiţi o mulţime minimă de cuie astfel încât fiecare scândură să fie bătută de cel puţin un cui. Se cere poziţia cuielor. *Formulat matematic*: găsiţi o mulţime de puncte de cardinal minim M astfel încât pentru orice interval $[a_i, b_i]$ din cele N, să existe un punct x din M care să aparţină intervalului $[a_i, b_i]$. Complexitate: O(N log N)

Exemplu:

- intrare: N = 5, intervalele: [0, 2], [1, 7], [2, 6], [5, 14], [8, 16]
- ieşire: M = {2, 14}
- explicaţie: punctul 2 se afla în primele 3 intervale, iar punctul 14 în ultimele 2

Soluţie: Se observa că dacă x este un punct din M care nu este capăt dreapta al nici unui interval, o translaţie a lui x la dreapta care îl duce în capătul dreapta cel mai apropiat nu va schimba intervalele care conţin punctul. Prin urmare, exista o mulţime de cardinal minim M pentru care toate punctele x sunt capete dreapta.

Astfel, vom crea mulţimea M folosind numai capete dreapta în felul următor:

- cât timp au mai rămas intervale nemarcate:
  - selectăm cel mai mic capăt dreapta, B_{min}; acesta trebuie să fie în M, deoarece este singurul punct care se afla în interiorul intervalului care se termină în B_{min}
  - marcăm toate intervalele nemarcate care conţin B_{min}
  - adăugăm B_{min} la M

Pentru a obţine o complexitate redusă, sortăm iniţial toate cele 2N capete şi le parcurgem de la stânga la dreapta. Pentru fiecare punct distingem cazurile:

- dacă este capăt stânga, introducem intervalul în lista de „intervale în procesare” şi trecem mai departe;

- dacă este capăt dreapta și intervalul respectiv nu conține nici un punct din M, atunci am găsit cel mai mic capăt dreapta al unui interval nemarcat, introducem capătul în M și marcăm toate intervalele din lista de intervale în procesare;
- dacă este capăt dreapta și intervalul din care face parte este deja marcat, trecem mai departe.

Complexitate:

- sortare: O(N log N)
- parcurgerea capetelor: O(N)
- adăugarea și ștergerea unui interval din lista de intervale în procesare: O(1)
- total: O(N log N)

3.2 Programare Dinamică

Programare dinamică presupune rezolvarea unei probleme prin descompunerea ei în subprobleme şi rezolvarea acestora. Spre deosebire de divide et impera, subproblemele nu sunt disjuncte, ci se suprapun.

Pentru a evita recalcularea porțiunilor care se suprapun, rezolvarea se face pornind de la cele mai mici subprobleme şi folosindu-ne de rezultatul acestora calculăm subproblema imediat mai mare. Cele mai mici subprobleme sunt numite subprobleme unitare, acestea putând fi rezolvate într-o complexitate constantă, ex: cea mai mare subsecvență dintr-o mulțime de un singur element.

Pentru a nu recalcula soluțiile subproblemelor ce ar trebui rezolvate de mai multe ori, pe ramuri diferite, se reține soluția subproblemelor folosind o tabelă (matrice uni, bi sau multi-dimensională în funcție de problemă) cu rezultatul fiecărei subprobleme. Aceasta tehnica se numește **memorizare**.

Aceasta tehnică determina ”valoarea” soluției pentru fiecare din subprobleme. Mergând de la subprobleme mici la subprobleme din ce în ce mai mari ajungem la soluția optimă, la nivelul întregii probleme. Motivul pentru care aceasta tehnica se numește Programare Dinamică este datorată flexibilității ei, ”valoarea” schimbându-și înțelesul logic de la o problema la alta. În probleme de minimizarea costului, ”valoarea” este reprezentata de costul minim. In probleme care presupun identificarea unei componente maxime, ”valoarea” este caracterizată de dimensiunea componentei.

După calcularea valorii pentru toate subproblemele se poate determina efectiv mulțimea de elemente care compun soluția. „Reconstrucția” soluției se face mergând din subproblemă în subproblemă, începând de la problema cu valoarea optimă și ajungând în subprobleme unitare. Metoda și recurența variază de la problemă la problemă, dar în urma unor exerciții practice va deveni din ce în ce mai facil să le identificați.

Aplicând aceasta tehnică determinăm **una** din soluțiile optime, problema putând avea mai multe soluții optime. În cazul în care se dorește determinarea tuturor soluțiilor optime, algoritmul trebuie combinat cu unul de backtracking în vederea construcției soluțiilor.

Aplicarea acestei tehnici de programare poate fi descompusă în următoarea secvență de pași:

1. Identificarea structurii și a metricilor utilizate în caracterizarea soluției optime;

2. Determinarea unei metode de calcul recursiv pentru a afla valoarea fiecărei subprobleme;
3. Calcularea "bottom-up" a acestei valori (de la subproblemele cele mai mici la cele mai mari);
4. Reconstrucția soluției optime pornind de la rezultatele obținute anterior.

3.2.1 Exemple de probleme

Programarea Dinamică este cea mai flexibilă tehnica din programare. Cel mai uşor mod de a o înțelege presupune parcurgerea cât mai multor exemple.

O problema clasică de Programare Dinamică este determinarea celui mai lung subşir strict crescător dintr-un şir de numere. Un subşir al unui şir este format din caractere (nu neapărat consecutive) ale şirului respectiv, în ordinea în care acestea apar în şir.

Exemplu: pentru şirul 24 12 15 15 8 19 răspunsul este şirul 12 15 19

Se observa că dacă încercăm o abordare greedy nu putem stabili nici măcar elementul de început într-un mod corect. Totuşi, problema se poate rezolva ”muncitoreşte” folosind un algoritm care alege toate combinațiile de numere din şir, validează că şirul obținut este strict crescător şi îl reține pe cel de lungime maximă, dar aceasta abordare are complexitatea temporală $O(2^N)$. Cu optimizări este posibil sa se ajungă la O(N!).

O metoda de rezolvare mai eficientă foloseşte *Programarea Dinamică*. Începem prin a stabili pentru fiecare element lungimea celui mai lung subşir strict crescător care începe cu primul element şi se termină în elementul respectiv. Numim aceasta valoare $best_i$ şi aplicăm formula recursivă $best_i = 1 + max(best_j)$, cu $j<i$ şi $elem_j < elem_i$.

Aplicând acest algoritm obținem:

```
elem 24 12 15 15 8 19

best  1  1  2  2 1  3
```

Pentru 24 sau 12 nu există nici un alt element în stânga lor strict mai mici decât ele, de aceea au best egal cu 1. Pentru elementele 15 se poate găsi în stânga lor 12 strict mai mic decât ele. Pentru 19 se găseşte elementul 15 strict mai mic decât el. Cum 15 deja este capăt pentru un subşir soluție de 2 elemente, putem spune că 19 este capătul pentru un subşir soluție de 3 elemente.

Cum pentru fiecare element din mulțime trebuie să găsim un element mai mic decât el şi cu best maxim, avem o complexitate O(N) pentru fiecare element. În total rezultă o complexitate $O(N^2)$. Se pot obține şi rezolvări cu o complexitate mai mica folosind structuri de date avansate. Atât soluția în $O(N^2)$, cât si o soluție în O(N log N) poate fi găsită la [5]. Tot acolo se poate găsi şi o lista de probleme mai dificile ce folosesc tehnica Programării Dinamice, adaptată în diferite forme.

Pentru a găsi care sunt elementele ce alcătuiesc subşirul strict crescător putem să reținem şi o „cale de întoarcere”. Reconstrucția astfel obținută are complexitatea O(N). Exemplu: subproblema care se termina în elementul 19 are subşirul de lungime maximă 3 şi a fost calculată folosind subproblema care se termină cu elementul 15 (oricare din ele). Subşirul de lungime maximă care se termină în 15 a fost calculat folosindu-ne de elementul 12. 12 marchează sfârşitul reconstrucției fiind cel mai mic element din subşir.

O altă problemă cu o aplicare clasică a Programării Dinamice este şi determinarea celui mai lung subșir comun a două șiruri de caractere. Descrierea problemei, indicații de rezolvare şi o modalitate de evaluare a soluțiilor voastre poate fi găsită la [6].

O problema care admite o varietate mare de soluții este cea a subsecvenței de sume maxime. Enunțul poate fi găsit la [7].

4 Concluzii şi observații

Aceste 2 tehnici sunt flexibile şi simpliste la nivel conceptul, dar cu ajutorul lor se pot rezolva probleme foarte complexe. În viitor este posibil să întâlniți algoritmi de Programare Dinamică pe arbori sau Greedy pe stări, unde fiecare stare este o matrice. Conceptele rămân neschimbate.

Aspectul cel mai important de reținut este că soluțiile găsite trebuie să reprezinte optimul global şi nu doar local. Se pot confunda uşor problemele care se rezolvă cu Greedy cu cele care se rezolvă prin Programare Dinamică.

5 Referințe

[1] http://en.wikipedia.org/wiki/Greedy_algorithm

[2] http://en.wikipedia.org/wiki/Dynamic_programming

[3] http://ww3.algorithmdesign.net/handouts/Greedy.pdf

[4] http://ww3.algorithmdesign.net/handouts/DynamicProgramming.pdf

[5] http://infoarena.ro/problema/scmax

[6] http://infoarena.ro/problema/cmlsc

[7] http://infoarena.ro/problema/ssm

[8] Capitolul IV din Introducere în Algoritmi de către T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein