Range minimum query

Apostu Croitoru Diana

Universitatea Politehnica Bucuresti
Facultatea de Automatica si Calculatoare
Grupa 321CA

Scopul lucrarii

Problema Range Mininum Query a fost rezolvata folosind diferiti algoritmi, implicit *Square root decomposition*, *Sparse Table* si *Segment Tree*. Scopul lucrarii este de a compara algoritmii folositi din doua puncte de vedere:
⇒ **cel al eficientiei temporale**
⇒ **al memoriei folosite**

1 Descrierea problemei rezolvate

1.1 Explicarea problemei

Scopul algoritmului Range minimum query este de a preciza elementul minim dintr-un vector,intre doua pozitii date. Se va urmari o implementare eficienta din punct de vedere al memoriei folosite si al timpului de executie,folosind diferiti algoritmi,comparand astfel complexitatea fiecarui algoritm.

1.2 Aplicație practică

Problema gasirii strămoșului comun cel mai apropiat ("LCA") este in stransa legatura cu utilizarea RMQ. Problema LCA poate fi redusă la o versiune restricționată a unei probleme RMQ, în care elementele consecutive din vector diferă exact cu 1.
RMQ-urile nu sunt utilizate doar cu LCA. Acestea au un rol important în preprocesarea șirurilor, unde sunt utilizate cu vectori de sufixe.De asemenea,LCA este folosit in compilatoare,ajutand la economisirea spatiului utilizat,dar si la eficientizarea codului.

2 Specificarea soluțiilor alese

Problema RMQ are multiple moduri in care se poate rezolva:
- solutia simpla de parcurgere a intervalelor si determinare a minimului

- folosind "Sparse Table" , ce raspunde fiecarei interogari in O(1) ,folosind o structura simpla si avand un timp de complexitate eficient

- folosind Arbori de intervale ("Segment Trees")

- Sqrt-decomposition ce raspunde la fiecare interogare in $O(\sqrt{N})$

- folosind "Cartesian Tree" , "Farach-Colton" si "Bender algorithm", ajungand la problema LCA ,implicit la un RMQ restrans ce utilizeaza proprietatile unui "Cartesian Tree"
Din soluțiile prezentate am ales să implementez următorii algoritmi:
⇒ **Sqrt Decomposition**
⇒ **Segment Trees**
⇒ **Sparse Table**

Se va realiza o descriere amanuntita a algoritmilor din urmatoarele puncte de vedere:
-cel al complexitatii temporale
-cel al complexitatii spatiale

3 Criterii de evaluare pentru soluția propusă

Pentru o verificare coerenta a algoritmilor , acestia se vor rula pe o multitudine de teste diversificate,restranse,avand un numar mic de interogari.De asemenea,corectitudinea algoritmilor poate fi verificata si din punct de vedere teoretic.
Eficienta algoritmilor va fi evidentiata printr-o serie de teste ce presupune un numar mare de interogari.Arborii de intervale sunt structuri de date de tip heap, care pot fi utilizate pentru efectuarea operațiunilor de actualizare/ interogare la intervale de matrice în timp logaritmic. Prin urmare acest algoritm se va remarca din punct de vedere al eficientei. De asemenea,testele in care elemetele din vector se pot modifica intre interogari reprezinta alt avantaj al acestui algoritm.Daca vectorul contine un numar mare de elemente,complexitatea primului algoritm este mare.

4 Prezentarea soluțiilor

4.1 Descrierea modului în care funcționează algoritmii aleși

Se va realiza o analiza asupra algoritmilor implementati , explicand cum functioneaza fiecare dintre ei :

Sparse Table

Sparse Table este un algoritm usor de implementat , fiind utilizat pentru interogări rapide pe un set de date statice (elementele nu se schimbă). Preprocesarea valorilor adaugate in matricea Sparse Table , permit raspunderea intr-un timp eficient la interogari.

Matricea Sparse Table se va precalcula in urmatorul mod:
SparseTable[i][j] = pozitia valorii din intervalul care incepe pe pozitia i si are lungimea 2^j
Solutiile se vor calcula progresiv (dinamic), pornind de la valorile obtinute pe intervale mici , ajungand la intervale din ce in ce mai mari. Matricea va fi prelucrata prin calcularea rezultatelor pentru fiecare pozitie posibila de plecare.
Fie vector v definit mai jos, si o interogare de tipul (stanga , dreapta).

7	2	3	0	5	10	3	12	18
a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8

Valoarea x = (int)log2(right left + 1) este folosita pentru a putea cuprinde intervalul (stanga, dreapta) prin doua intervale .Reuniunea acestora este chiar intervalul (stanga,dreapta). Unul dintre acestea incepe pe pozitia din stanga si are lungimea 2^x si unul care incepe pe pozitia dreapta 2^x+1 si se termina pe pozitia dreapta. Lungimea celor doua intervale corespunde(2^x), deci valoarea minima este precalculata ın SparseTable.
Raspunsul la interogare va :
min(v[SparseTable[stanga][2^x]], v[SparseTable[dreapta - 2^k+1][dreapta]).
De exemplu, pentru vectorul dat , minimum pentru o interogare folosind SparseTable ar fi :
Minimum pentru interogarea [4,7] este 3
Minimum pentru interogarea [0,4] este 0
Minimum pentru interogarea [7,8] este 12

Segment Trees (Arbori de intervale)

Arborele de intervale este structura folosita pentru rezolvarea acestui algoritm. Un arbore de intervale este un arbore binar in care fiecare nod poate avea asociata o structura auxiliara (anumite informatii) ,implicit in problema aleasa, minimul dintr-un subinterval al vectorului dat.

Dandu-se doua numere intregi st si dr, cu $st < dr$, atunci arborele de intervale T(st, dr) se construieste recursiv astfel:
- consideram radacina nod avand asociat intervalul [st, dr];
- daca $st < dr$ atunci vom avea asociat subarborele stang T(st, mij), respectiv subarborele drept T(mij + 1, dr), unde mij este mijlocul intervalului [st, dr].

De exemplu, pentru un vector cu N elemente ,daca nodul curent contine minimum din intervalul [1,N] atunci copilul drept va contine minimum din intervalul (N / 2 + 1, N) , implicit copilul stang va contine minimum din intervalul (1, N / 2).
Implementarea cea mai simpla este cea recursiva , impartindu-se mereu intervalul in doua subintervale egale , care sunt preprocesate separat.
Asupra unui arbore de intervale se pot face doua operatii semnificative: actualizarea(Context), respectiv interogarea unui interval(Find).Cea mai eficienta metoda de stocare in memorie a unui arbore de intervale este sub forma unui vector folosind aceeasi codificare a nodurilor precum la heap-uri.Rezultatele se vor actualiza succesiv, ajungand astfel la rezultatul dorit.

Sqrt Decomposition

Algoritul Sqrt Decomposition are ca principiu de baza preprocesarea .De exemplu, daca avem un vector cu N elemente, acesta se va imparti in intervale aproape egale(blocks) de dimensiune $\sqrt{N}$.Este posibil ca ultimul interval sa aiba mai putine elemente, dar nu este o problema, fiind rezolvat usor. Pentru fiecare block vom calcula minimul din intervalul respectiv, astfel valoarea lui va fi adaugata in block-ul sau. O abordare ineficienta este pur și simplu de a itera prin fiecare element din intervalul l pentru a calcula răspunsul corespunzător. Prin urmare, complexitatea de timp pe interogare va fi O (n). Solutia algoritmului Sqrt Decomposition este de a parcurge vectorul sarind $\sqrt{N}$ pasi , pentru fiecare block in care se afla minimum din subinterval, ajungand la raspunsul dorit pentru intervalul I.

3 5 2 4 6 0 3 5 1

block[0] block[1] block[2]
2 0 1

De exemplu, pentru o interogare (3,8) valoarea minimului va fi 0. Se va parcurge iterativ block[1] si block[2] , ajungand de rezultat. Daca vom avea o interogare (2,8) se va parcurge elementul cu indicele 2 din vector , apoi block[1], block[2]. Aceasta rezolvare va diminua semnificativ numarul de operatii efectuate.

4.2 Analiza complexității soluțiilor

Sparse Table

Se citesc elementele din vector,se inițializeaza prima linie din SparseTable și se declara vectorul de logaritm precalculate. Complexitatea de timp va fi constituita de while-uri ,deoarece opertiile se efectuează ın timp constant . Așadar, pentru citirea datelor din fisier , complexitatea va fi O(n).

```
30      ofstream g;
31      f.open(argv[1]);
32      g.open(argv[2]);
33
34      int N;
35      int M;
36      f >> N >> M;
37
38      vector<vector<int>> SparseTable;
39
40      int value;
41      vector<int> logaritm, v;
42
43      logaritm.push_back(-1);
44      int p  = 1;
45      while(p <= N) {
46          logaritm.push_back(logaritm[p / 2] + 1);
47          p++;
48      }
49      SparseTable.assign(N, vector<int>(logaritm[N] + 1));
50      int i = 0;
51      while(i < N) {
52          f >> value;
53          v.push_back(value);
54          SparseTable[i][0] = i;
55          i++;
```

Functia compute realizeaza prelucrarea Matricei SparseTable , folosind doua while-uri. Complexitatea de timp va fi constituita de while-uri , deoarece operatiile se efectueaza in timp constant. Se observa ca primul while va efectua log(N) operatii,iar cel de-al doilea N + 1 - 2^j. Complexitatea totala este data de numarul de operatii efectuate .

$Nr.operatii = \sum_{j=1} N + 1 - 2^j$ = NlogN - 2N

Complexitatea de timp a functiei Compute este O(NlogN).

```
void Compute(int N, vector<int> v, vector<vector<int>> &SparseTable) {
    int j = 1;
    while((1 << j) <= N){
        int i = 0;
        int aux;
        while(i + (1 << j) - 1 < N) {
            if (v[SparseTable[i][j - 1]] > v[SparseTable[i + (1 << (j - 1))][j - 1]]) {
                aux = SparseTable[i + (1 << (j - 1))][j - 1];
                SparseTable[i][j] = aux;
            } else {
                SparseTable[i][j] = SparseTable[i][j - 1];

            }
            i++;
        }
        j++;
    }
}
```

Preprocesarea fiecarei interogari se va realiza in O(1).Asadar,daca am avea X interogari, complexitatea va fi O(X) .

```
    int left;
    int right;
    int j = 0;
    while(j< M) {
        f >> left >> right;

        int lg = logaritm[right - left + 1];
        int solution;
        if (v[SparseTable[left][lg]] < v[SparseTable[right - (1 << lg) + 1][lg]])
            {
                solution = v[SparseTable[left][lg]];
            }
        else {
            solution =  v[SparseTable[right - (1 << lg) + 1][lg]];
        }

        g << solution << "\n";
        j++;
    }


    f.close();
    g.close();
    return 0;
}
```

Raspunsul este calculat prin compararea a doua numere din SpaseTable. Rezultatul va fi scris in fisierul de iesire(g). Complexitatea generala a algoritmului SpaseTable este O(NlogN) , iar complexitatea de memorie coincide cu cea generala.

Sqrt Decomposition

Se construieste vectorul , citind valorile de la tastatura.Complexitatea de timp va fi constituita de un for,deoarece operatiile se efectueaza in timp constant.Pentru citirea datelor din fisier , complexitatea va fi O(n). Se apeleaza functia de preprocesare, care construieste blocurile de dimensiune $\sqrt{N}$.

```
58  int main(int argc,
59    const char * argv[]) {
60
61    ifstream f;
62    ofstream g;
63    int stanga, dreapta;
64    f.open(argv[1]);
65    g.open(argv[2]);
66    int x;
67    int y;
68    int value;
69    f >> x >> y;
70    for (int i = 1; i <= x; i++) {
71      f >> value;
72      arr[i] = value;
73    }
74    process(arr, x, g);
```

Functia de preprocesare are complexitate O(n). Functia va calcula numarul de blocuri, apoi pentru fiecare bloc , se va parcurge subintervalul de dimensiune $(blk_(sz))$,completandu-se fiecare bloc cu minimum sau. Complexitatea proprocesarii este data de un while si un for, dar aceasta este relativ slaba in comparatie cu algoritmul alterior.(O(N))

```
void process(int input[], int n, ofstream & g) {
  int blk_idx = 0;
  blk_sz = sqrt(n);
  long long min = 9223372036854775807;
  int size = 0;
  int p = 0;
  while (p <= n) {
    min = 9223372036854775807;
    for (int j = p; j < p + blk_sz; j++) {
      if (min > input[j] && input[j] != 0) {
        block[blk_idx] = input[j];
        min = input[j];
      }
    }
    blk_idx++;
    size++;
    p = p + blk_sz;
  }

}
```

Raspunsul pentru ecare interogare se face in O($\sqrt{N}$). Functia query implementeaza cautarea minimului pentru o interogare (stanga, dreapta) astfel : se va parcurge intai subsirul din partea stanga a vectorului,apoi subsirul dat de blocurile realizate , si in cele din urma subsirul din partea dreapta. Complexitatea este data de cele 3 while-uri , deoarece operatiile se efectueaza in timp constant. Complexitatea functiei query este implicit O($\sqrt{N}$).

```
int blk_sz;

int query(int l, int r, ofstream & g, int arr[]) {
  int min = 2147483647;
  while (l <= r and l % blk_sz != 0 and l != 0) {
    if (min > arr[l]) {
      min = arr[l];
    }
    l++;
  }
  while (l + blk_sz <= r) {
    if (min > block[l / blk_sz]) {
      min = block[l / blk_sz];
    }
    l = l + blk_sz;
  }
  while (l <= r) {
    if (min > arr[l]) {
      min = arr[l];
    }
    l++;
  }
  return min;
}
```

Complexitatea de memorie a algoritmului este O(N).

Segment Trees

Algoritmul Segment Trees se bazeaza pe doua functii de baza, de actualizare(Context) si interogarea unui interval (Find).
Se vor citi datele de intrare , iar ,pentru fiecare valoare se va apela functia de Context, actualizand minimurile din arbore,punand valoarea citita pe pozitia

corespunzatoare.Complexitatea de timp este O(n).

```
55  int main(int argc,
56    const char * argv[]) {
57    ifstream f;
58    ofstream g;
59    g.open(argv[2]);
60    f.open(argv[1]);
61
62    int x;
63    int y;
64    int i;
65    i = 1;
66    f >> x >> y;
67    vector < int > Segments;
68    int data;
69    Segments.reserve(4 * x);
70    while (i <= x) {
71      f >> data;
72      Context(i, data,Segments, 1, 1, x);
73      i++;
74    }
```

Fiecare apel a operatiei Context este realizat in O(1) , deoarece este in timp constant, iar numarul total de niveluri al arborelui(inaltimea) este log(N) . Asadar,complexitatea functiei Context este de O(logn).Functia Context proceseaza un singur nod pe nivel, pornind de la 1 si ajungand la N .(fiecare

nod are 2 copii, copilul stang si copilul drept).

```
int getMid(int s, int e) {
  return s + (e - s) / 2;
}

void Context( int pos, int value,vector < int > & Segments, int node, int left, int right) {
  int aux;
  if (left == right) {
    aux = value;
    Segments[node] = aux;
    return;
  }
  int m = getMid(left, right);
  if (pos > m) {
    Context(pos, value,Segments,  2 * node + 1, m + 1, right);

  } else {
    Context(pos, value,Segments,  2 * node, left, m);
  }

  if (Segments[2 * node] < Segments[2 * node + 1]) {
    Segments[node] = Segments[2 * node];
  } else {
    Segments[node] = Segments[2 * node + 1];
  }
}
```

Functia Find este apelata in O(1) ,iar numarul de noduri procesate este maxim 2 pe nivel. Numarul de intervale este implicit log(N),iar operatiile realizate sunt in timp constant.Complexitatea functiei Find este de O(logN).

```
void Find(int start, int end, vector < int > & Segments, int node, int left, int right, int & Minimum) {
  int ok = verify(start, right, end, left);
  if (ok) {
    if (Segments[node] < Minimum) {
      Minimum = Segments[node];
    }
    return;
  }
  int m = getMid(left, right);
  if (m < end) {
    Find(start, end, Segments, 2 * node + 1, m + 1, right, Minimum);
  }
  if (start <= m) {
    Find(start, end, Segments, 2 * node, left, m, Minimum);
  }
}
```

Functia Find va fi apelata pentru fiecare interogare citita. Pentru functia Find complexitatea acesteia este de log(N). Daca exista M interogari , atunci complexitatea de timp este de O(M * log(N)) .

Deoarece este nevoie de un vector ce retine 4 * N elemente , complexitatea de memorie va fi O(N).

```
int position, right, left, Minimum;
int j = 0;
bool type;
while (j < y) {
  bool type = false;
  if (type == false) {
    f >> right >> left;
    Minimum = MaxValue;
    Find(right + 1, left + 1, Segments, 1, 1, x, Minimum);
    g << Minimum << "\n";
  }
  j++;
}
```

4.3 Prezentarea principalelor avantaje și dezavantaje pentru soluțiile luate în considerare

Implementarea cu Sparse Table

Avantaje:
- raspunde la intrebari in O(1)
- are o structura simpla
- complexitatea termporala este foarte eficienta

Dezavantaje:
- complexitatea de memorie este de O(NlogN)
- la schimbarea unui element din vector este nevoie de refacerea intregii structuri de date
- complexitatea de prelucrare este O(NlogN)

Implementarea cu Sqrt Decomposition

Avantaje:
- are o structura simpla
- implementare scurta ,rapida

Dezavantaje:
- are complexitatea slaba(complexitatea pentru prelucrare este O(N))
- complexitatea pentru fiecare interogare are O($\sqrt{N}$)

Implementarea cu Segment Trees

Avantaje:
- se pot face modificari asupra vectorului de input(O(N))
- complexitatea de memorie este eficienta(O(N))
- implementare scurta , rapida

Dezavantaje:
- raspunde la fiecare interogare in O(logN)
- intelegerea conceptelor structurale ,utilizate in rezolvarea algoritmului

5 Evaluare

5.1 Descrierea modalitatii de construire a setului de teste folosite pentru validare

Generarea testelor a fost realizata pentru toti algoritmii, folosind valori mari ale elementelor din vectorul de input. In folderul "in" exista 10 fisiere de input de dimensiuni diferite :
- primele 4 au numarul de interogari mai mic de 50
- urmatoarele 5 cu numarul de interogari mai mic de 2000
- ultimul are dimensiunea cea mai mare <= 7000 interogari

De asemenea, deoarece niciun algoritm nu depinde de dimensiunea valorilor din vectorul dat , testele din folderul other.tests au numarul de interogari cuprins intre 2000 si 6000, respectiv un numar mare de elemente in vector, cu valori mari(3200 < x < 7800). Pentru verificarea algoritmilor, am generat raspunsurile aplicand algoritmul cel mai simplu, comparandu-le apoi intre ele. Numerele alese pentru generarea testelor sunt de dimensiuni mari,implicit si dimensiunea vectorului, prin urmarea construirea testelor a fost realizata folosind un generator de teste.

5.2 Menționați specificațiile sistemului de calcul pe care ați rulat testele

Procesor : Intel Core i7-7700HQ CPU @2.80 GHz,2.80 GHz
Memorie RAM : 8 GB

5.3 Ilustrarea rezultatelor evaluarii pe setul de teste

Eficienta temporala (s)					
Nr.test	Nr.elemente vector	Nr.interogari	Sparse Table	Segment Tree	Sqrt Decomposition
1	34	31	0.048	0.045	0.251
2	22	92	0.184	0.235	0.388
3	57	9	0.036	0.044	0.228
4	56	34	0.053	0.049	0.157
5	778	208	0.518	0.390	0.670
6	637	1989	3.769	4.902	4.108
7	2295	178	0.385	0.450	0.567
8	1303	1184	1.545	2.680	3.120
9	274	324	0.827	0.835	0.944
10	3599	6734	11.947	12.103	10.880
11	3111	2210	6.637	6.382	4.685
12	7608	5678	11.600	12.864	16.945

5.4 Explicarea rezultatelor evaluarii pe setul de teste

Din tabelul ilustrat mai sus, se poate observa ca SparseTable este cel mai eficient algortim dintre cei trei. In cazul testelor 11 si 5, se poate observa ca performanta algoritmului Segment Tree este mult mai buna ,in detrimentul lui SparseTable. Arborii de intervale sunt eficienti atunci cand exista un numar mic de interogari si foarte multe elemente in vector.(se vor realiza putine operatii pentru a construi arborele) De asemenea, se observa ca Sqrt Decomposition are o complexitate relativ slaba,deoarece eficienta sa temporala depinde de numarul de blocuri construite si parcurse. Performanta lui SparseTable este data de raspunsurile interogarilor, care se realizeaza in O(1). Asadar, arborii de intervale sunt mult mai eficienti doar in anumite

cazuri, fiind dependenti de datele de intrare: algoritmul raspunde la interogari mai greu, dar preproceseaza datele mai repede. Testele 10 si 11 pun in evidenta algoritmul Sqrt Decomposition . Acesta este eficient atunci cand se construiesc blocuri de dimensiuni mari ,iar interogarile se afla in intervale de dimensiune $\sqrt{N}$, ce se pliaza de blocurile construite. Cum algoritmul SparseTable realizeaza preprocesarea in O(NlogN), iar Sqrt in O(N), se poate intampla ca algoritmul Sqrt Decomposition sa fie mai eficient ,in functie de interogarile la care este nevoit sa raspunda.Complexitatea algoritmului variaza in functie de datele de input.

Bibliografie

1. InfoArena
https://infoarena.ro/problema/rmq
2. Standford Univeristy - Range minimum queries
https://web.stanford.edu/class/cs166/lectures/01/Slides01.pdf
3. Algorithmsandme
https://algorithmsandme.com/range-minimum-query-rmq/
4. Algorithms.com
https://cp-algorithms.com/sequences/rmq.html
5. ResearchGate.com Faster range minimum queries
6. Hackrearth
https://www.hackerearth.com/practice/notes/sparse-table/
7. Topcoder
https://www.hackerearth.com/practice/notes/sparse-table/