Folosirea arborilor de intervale pentru optimizarea problemelor pe grafuri

Alexandru Luchianov

Iulie 2021

Mulțumiri lui Tamio-Vesa Nakajima pentru revizuirea și editarea acestui articol.

1 Cuvânt înainte

Arborii de intervale[1] sunt o structură puternică cu utilizări neașteptate, iar cunoașterea acestora este necesară pentru a înțelege acest articol. De asemenea, mai sunt necesare noțiuni și algoritmi elementari de grafuri.

2 Problema "Legacy"

Enunț: Se dă un graf cu N ($N \leq 10^5$) noduri și M ($M \leq 10^5$) mănunchiuri de muchii. Mănunchiurile de muchii pot fi de 3 tipuri.

1. Există o muchie de la nodul X la nodul Y de cost Z;
2. Există muchii de la nodul X la nodurile $L, L+1, \ldots, R$ de cost Z;
3. Există muchii de la nodurile $L, L+1, \ldots R$ la nodul X de cost Z.

Se cere distanța de la nodul 1 la nodul N. Problema originală cu tot cu poveste se poate găsi la `codeforces.com/contest/786/problem/B`.

[1] infoarena.ro/problema/arbint

Soluția brută: O soluție brută ar consta în a adăuga toate muchiile generate de mănunchiuri și a folosi ulterior un algoritm standard de găsire a drumului minim in graf (de exemplu algoritmul lui Dijkstra). În cel mai rău caz, această soluție ar genera un graf cu NM muchii și astfel ar avea o complexitate totală de $\mathcal{O}(NM \log (NM))$

Soluția completă: Pentru a trata și cazurile în care apar și alte tipuri de mănunchiuri, avem următoarele variante:

- Folosim un algoritm specializat pentru această problemă *(adaptăm algoritmul la problemă)*;
- Construim un alt graf pe care folosim ca *blackbox*[2]un algoritm general de găsirea celui mai scurt drum în graf *(adaptăm problema la algoritm)*.

De multe ori este mult mai ușor și mai rapid să adaptăm problema la algoritm deoarece ne putem folosi de un *blackbox* și utiliza librării deja existente, însă, în multe cazuri, nu putem face acest lucru.

Pe scurt, obiectivul nostru este să transformăm graful dat prin mănunchiuri într-un graf cu cât mai puține muchii în care să se mențină distanțele dintre nodurile $1 \leq X, Y \leq N$ și pe care să folosim un algoritm binecunoscut. Vom numi nod *real* un nod din cele originare de la 1 la N și vom numi nod *auxiliar* orice alt nod.

Vom începe prin a construi pentru fiecare interval cu capetele între 1 și N și cu lungimea putere de 2 câte un nod auxiliar care să aibă drumuri de cost 0 la toate nodurile *reale* din interval și care să nu aibă drumuri la nodurile *reale* din afara intervalului. Astfel, vom construi $\mathcal{O}(N \log N)$ noduri auxiliare.

Pentru a construi în mod eficient toate nodurile auxiliare, le vom construi pe niveluri. La nivelul i vom construi nodurile auxiliare care reprezintă intervale de lungime 2^i. Se observă faptul că, atunci când construim un nod de pe nivelul i, putem ca, în loc de a-l conecta brut la nodurile din interval, să îl descompunem în 2 intervale de lungime 2^{i-1} și să îl conectăm la nodurile auxiliare care corespund acestor intervale.

Pentru a procesa mănunchiurile de tip 2 vom alege 2 noduri auxiliare astfel încât reuniunea intervalelor lor să fie $[L, R]$ și vom trage muchii de cost Z de la X la ele.

Un cititor experimentat poate observa faptul că rețeaua și construcția acesteia seamănă foarte mult cu structura RMQ-ului[3].

Se poate observa că mănunchiurile de tip 3 pot fi tratate în mod analog prin construcția unei rețele în care muchiile doar sunt orientate invers.

[2]un algoritm în interiorul căruia nu știm ce se întâmplă, dar știm ce îi putem da și ce ne va da înapoi

[3]`infoarena.ro/problema/rmq`

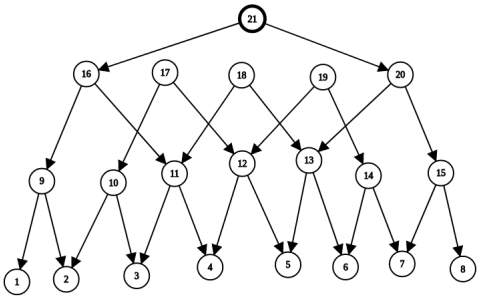

Figure 1: Nodurile auxiliare cu structura de tip RMQ

Astfel, am obținut o rețea cu $N \log N$ noduri și $2N \log N + 2M$ muchii, însă putem reduce numărul de noduri la doar N noduri auxiliare. Chiar dacă acest lucru nu este necesar pentru problema curentă, vom introduce un alt tip de rețea care o să fie folositoare în problemele următoare. Vom crea un nod auxiliar pentru intervalul $[1, N]$ de la care vom trage muchii către 2 intervale $\left[1, \frac{N}{2}\right]$ și $\left[\frac{N}{2} + 1, N\right]$ pentru care vom crea noduri auxiliare în mod recursiv. Astfel vom genera N noduri auxiliare în total. Pe scurt, creăm nodurile auxiliare așa cum am crea și nodurile unui arbore de intervale[4].

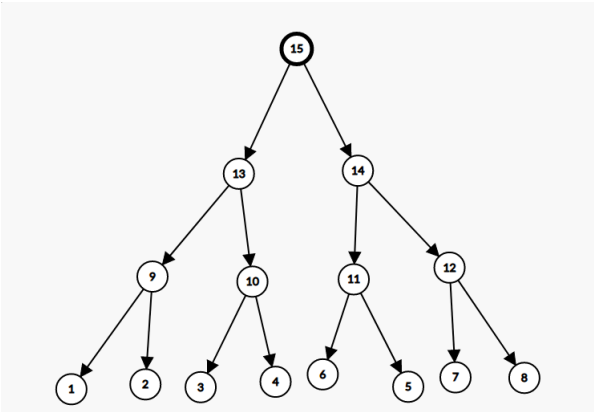

Figure 2: Nodurile auxiliare cu structura de tip AINT

[4]https://www.infoarena.ro/arbori-de-intervale

Mai jos se găsește o funcție care și construiește această structură (muchiile sunt înregistrate într-un vector de pair-uri numit `edges`):

```
void build(int node, int from, int to) {
  if(from < to) {
    int mid = (from + to) / 2;
    aint[node] = ++nodeids; // Dam un nume nefolosit nodului
    // auxiliar care acopera [from, to];
    //spargem in 2 intervale si construim recursiv.
    build(node * 2, from, mid);
    build(node * 2 + 1, mid + 1, to);
    //adaugam muchii catre fii
    edges.push_back({aint[node], aint[node * 2]};
    edges.push_back({aint[node], aint[node * 2 + 1]};
  } else
    aint[node] = from;
}
```

Procesarea unui mănunchi se face prin descompunerea intervalului $[L, R]$ în $\log N$ intervale din arborele de noduri auxiliare (exact ca la un arbore de intervale clasic). Mai jos se găsește și o implementare a unui cod care trage muchii de la nodul *target* la intervalul $[l, r]$:

```
void query(int node, int from, int to, int l, int r, int target) {
  if(from == l && to == r) {
    edges.push_back({target, aint[node]});
  } else {
    int mid = (from + to) / 2;
    if(l <= mid && r <= mid)
      query(node * 2, from, mid, l, r, target);
    else if(mid + 1 <= l && mid + 1 <= r)
      query(node * 2 + 1, mid + 1, to, l, r, target);
    else {
      query(node * 2, from, mid, l, mid, target);
      query(node * 2 + 1, mid + 1, to, mid + 1, r, target);
    }
  }
}
```

Astfel, obținem un graf cu N noduri auxiliare și $2N + 2M \log N$ muchii.

3 Problema "Small graph"

Enunț: Se dă o permutare de lungime N ($N \leq 1000$) și se cere un graf cu maxim $30N$ noduri și maxim $30N$ muchii cu proprietatea că pentru orice i și j astfel încât $1 \leq i, j \leq N$ și $i \neq j$ există drum de la i la j dacă și numai dacă $i < j$ și $p_i > p_j$. Problema originală cu tot cu poveste se poate găsi la `codeforces.com/gym/101382`

Soluție: Un avantaj al structurii de tip AINT este faptul că practic avem o structură *online* în care putem face modificări pe parcurs. De exemplu, am putea *activa* și *dezactiva* noduri astfel încât să existe drum de la nodul auxiliar care corespunde intervalului $[L, R]$ la nodul original i dacă și numai dacă $L \leq i \leq R$ și i este activ fără să trebuiască să recalculăm tot arborele de intervale.

Putem recalcula doar nodurile auxiliare cărora li s-a produs o schimbare în intervalul asociat în mod recursiv. Un cod care *activeaza* sau *dezactiveaza* un nod în mod recursiv ar fi:

```
void update(int node, int from, int to, int x, int active) {
  if(from < to) {
    int mid = (form + to) / 2;
    aint[node] = ++nodeids; // Redenumim nodul auxiliar,
    // recalculam recursiv intervalul in care s-a produs
    // schimbarea.
    if(x <= mid)
      update(node * 2, from, mid, x, active);
    else
      update(node * 2 + 1, mid + 1, to, x, active);
    // Conectam noul nod auxiliar la fiii sai.
    if(0 < aint[node * 2])
      edges.push_back({aint[node], aint[node * 2]};
    if(0 < aint[node * 2 + 1])
      edges.push_back({aint[node], aint[node * 2 + 1]};
  } else {
    if(active == 0)
      aint[node] = 0; // 0 reprezinta un nod null care nu exista.
    else
      aint[node] = from;
  }
}
```

Acum că am descoperit ce putem face cu adevărat cu această structură ne putem întoarce la problema originală. Inițial, să spunem că nu avem niciun nod activ. Parcurgem pozițiile din permutare în ordine crescătoare după P_i și adăugăm muchii la intervalul $[i + 1, N]$ folosind structura de noduri auxiliare (exact ca la problema anterioară), iar apoi activăm nodul curent. In momentul în care un nod este activat, toate intervalele care îl conțin trebuie recalculate folosind funcția de update.

4 Problema "Mike and code of permutation"

Enunț: *Codificarea* unei permutări P de lungime N se definește ca un șir A de lungime N. Inițial, niciun element nu este marcat. Șirul A este construit de la stânga la dreapta aplicând următorul procedeu pentru fiecare i de la 1 la N:

- Se găsește j minim astfel încât j nu este marcat și $P_i < P_j$;
- Dacă există un astfel de j, atunci A_i devine j, iar j devine marcat;

- Dacă nu există un astfel de j, atunci A_i devine -1.

Se dă codificarea unei permutări și se cere o permutare care să corespundă acestei codificări. Se garantează că există cel puțin o permutare care să aibă această codificare. Problema originală cu tot cu poveste se poate găsi la: `codeforces.com/contest/798/problem/E`

Soluție: Pentru o codificare A definim C_i ca momentul în care a fost marcat i. In mod matematic:

$$C_i = \begin{cases} x & \text{dacă există } A_x = i, \\ n+1 & \text{altfel.} \end{cases}$$

Din A și C obținem următoarele informații despre vectorul P:

1. Pentru orice i cu $A_i = -1$ știm că toate elementele care au fost marcate dupa momentul i sunt mai mici decât elementul i. In mod formal: $P_i > P_j$ pentru orice j cu $i < C_j$;
2. Pentru orice i cu $1 \leq A_i$ știm că toate elementele la stânga lui A_i care au fost marcate după momentul i sunt mai mici decât elementul i. In mod formal: $P_i > P_j$ pentru orice j cu $j < A_i$ si $i < C_j$;
3. Pentru orice i cu $C_i \leq N$ știm că $P_i > P_{C_i}$.

Aceste condiții deduse din A și C sunt necesare și suficiente pentru ca o permutare să aibă codificarea A. Dacă ar fi să adăugăm fiecare condiție de mai sus în mod separat, am obține N^2 condiții de forma $P_X > P_Y$ pentru anumite perechi de forma (X, Y).

Pentru a rezolva aceste condiții putem construi un graf cu n noduri unde fiecare condiție de forma $P_X > P_Y$ e interpretată ca o muchie de la x la y. Apoi, vom sorta nodurile acestui graf în ordine topologică[5] pentru a obține o posibilă sortare descrescătoare după P_i a pozițiilor din permutare.

Am putea reduce numărul de muchii la aproximativ $N \log N$ folosind trucul prezentat în problemele anterioare, însă chiar dacă am face asta, problema tot nu ar fi rezolvată deoarece nu ne-am încadra în limita de memorie, așa că trebuie să găsim o metodă prin care nu trebuie să reținem graful în mod explicit. O variantă posibilă pe care am menționat-o și mai sus, ar fi să "adaptăm algoritmul la problemă" și nu "problema la algoritm" așa cum am făcut la ultimele 2 probleme.

Pentru a reuși să modificăm "algoritmul la problemă" trebuie mai întâi să înțelegem cum funcționează sortarea topologică și nu o mai putem trata drept blackbox. Pe scurt, algoritmul de sortare topologică pornește de la un nod nevizitat, îi vizitează în mod recursiv vecinii care încă nu au fost vizitați, iar apoi nodul curent este adăugat la începutul șirului generat de sortarea topologică. Bottleneck-ul constă în extragerea vecinilor nevizitați ai unui nod. Mai întâi

[5] infoarena.ro/problema/sortaret

vom verifica dacă nodul C_i este nevizitat, apoi îl vizităm și parcurgem în mod recursiv în continuare, dacă este nevoie.

Să presupunem că $1 \leq A_i$ (cazul alternativ în care $A_i = -1$ poate fi tratat ca și cum A_i ar fi $N + 1$). Vecinii sunt nodurile $j(1 \leq j \leq A_i - 1)$ cu $i < C_j$. Extragem nodul nevizitat $j(1 \leq j \leq A_i - 1)$ cu cel mare C_j. Dacă $C_j \leq i$, atunci ne oprim deoarece este clar că nu există un alt nod care să respecte condițiile, altfel îl marcăm pe j ca vizitat și îl parcurgem recursiv, apoi alegem un alt j cu C_j maxim din nodurile nevizitate rămase.

Singura dificultate rămasă este cum să implementăm extragerea elementului nemarcat cu valoarea maximă dintr-un interval. Acest lucru se poate implementa folosind arbori de intervale: când marcăm un element, pur și simplu îi schimbăm valoarea asociată în -1 ca să nu mai fie selectat mai târziu.

5 Problema "Transform"

Enunț: Se dau N, Q și două șiruri L și R indexate de la $N+1$ la $N+Q$ astfel încât $L_{i-1} \leq L_i$ și $R_{i-1} \leq R_i$. Se cere să se construiască un graf cu maxim $4N+2Q$ noduri astfel încât să existe drum de la un nod Y $(N+1 \leq Y \leq N+Q)$ la X $(1 \leq X \leq N)$ dacă și numai dacă $L_Y \leq X \leq R_Y$.

Problema originală cu tot cu poveste se poate găsi la: `infoarena.ro/problema/transform3`

Soluție: Un prim gând ar fi să încercăm să construim o rețea de tip RMQ peste nodurile de la 1 la N și să o folosim pe ea ca să tragem muchiile, însă așa am obține $2N \log N + 2Q$ muchii, ceea ce este prea mult. O altă posibilă idee ar fi să încercăm construirea unui rețele de tip AINT, dar aceasta ar obține $2N + 2Q \log N$ muchii, ceea ce tot nu s-ar încadra.

Arborii de intervale nu ar fi avut nevoie de condiția $L_{i-1} \leq L_i$ și $R_{i-1} \leq R_i$, așa că, cel mai probabil, soluția optimă se folosește de aceste restricții pentru a obține un număr mic de muchii. De aici putem deduce faptul că setul de noduri $[L_i, R_i]$ se comportă ca o coadă unde se inserează elemente într-un capăt și se șterg elemente de la celălalt. Dacă am putea crea o structură de date cu următoarele operații, am rezolva problema:

- Se inserează un nou nod în față;
- Se șterge un nod din spate;
- Se cere un nod auxiliar care să aibă drumuri numai către nodurile din structură.

Insă o astfel de structura ar fi greu de implementat, așa că ar fi benefic ca mai întâi să implementăm o structura mai simplă. Un *truc* oarecum cunoscut ar fi să implementăm o coadă prin 2 stive în următorul fel:

- Vom numi stivele S_1 și S_2.

- Elementele cozii (de la cele mai recent la cele mai târziu adăugate) sunt formate din elementele lui S_1 (de la vârf la bază), urmate de elementele lui S_2 (de la bază la vârf).
- Pentru o operație de inserare în coadă, inserăm în S_1.
- Pentru o operație de ștergere din coadă:
  - Dacă S_2 are cel puțin un element, îi vom elimina vârful.
  - Altfel, înseamnă că S_2 e goală, așa că trebuie ca mai întâi să transferăm toate nodurile din S_1 în S_2. Acest lucru se poate face prin eliminarea succesivă a vârfului lui S_1 și inserarea acestuia în S_2. Acum că S_2 are cel puțin un element, îi putem elimina vârful.

Astfel, fiecare element este inserat sau șters din ambele stive exact o singură dată. Dacă am reuși să implementăm o structură de tip stivă, am putea implementa și structura de tip coadă. Structura de tip stivă ar trebui să accepte următoarele operații:

- Se inserează un element în vârful stivei.
- Se șterge nodul din vârful stivei.
- Se cere un nod auxiliar care să aibă drumuri numai către nodurile din structură.

Pentru a implementa această structură vom menține o listă de noduri auxiliare și vom proceda în felul următor:

- Când avem de inserat un nou nod X, creăm un nod auxiliar Y, tragem muchii de la Y la X și de la Y la primul element din listă și adăugăm pe Y la începutul listei.
- Când avem de șters vârful stivei, eliminăm primul element din lista de noduri auxiliare.
- Nodul auxiliar cu drumuri numai către nodurile din structura este chiar primul nod din listă.

In concluzie, dacă implementăm coada folosind 2 stive, vom obține $2 \cdot 2 \cdot N$ muchii auxiliare pentru inserări (de fiecare dată când inserăm într-o stivă, creăm 2 muchii, iar fiecare element este inserat în ambele stive) și încă $2Q$ muchii pentru conectarea fiecărui nod de la $N+1$ la $N+Q$ la vârfurile ambelor stive. In total obținem $= 4N + 2Q$ muchii.

6 Alte probleme

- codeforces.com/problemset/problem/1158/C
- atcoder.jp/contests/abc210/tasks/abc210_f
- szkopul.edu.pl/problemset/problem/xNjwUvwdHQoQTFBrmyG8vD1O/site/?key=statement
- atcoder.jp/contests/arc069/tasks/arc069_d
- codeforces.com/problemset/problem/786/E