DSU с триене

Основна задача

Основната задача, която тази лекция ще реши, е следната: Първоначално имаме върхове без ребра. Имаме q на брой заявки. Има 3 типа заявки:

- **add** u v**:** Добавя се ново ребро в графа.
- **rem** u v**:** Изтрива се ребро от графа.
- **conn** u v**:** Да се изведе YES или NO в зависимост от това дали има път между u и v.

Преговор на DSU

DSU се състои от 2 функции:

- root(int x), която намира представителя на компонентата, в която се намира x.
- union(int u, int v), която обединява компонентите, в които се намират u и v.

За да правим тези операции бързо, използваме 2 оптимизации:

- Компресираме пътя между x и представителя на компонентата. По този начин създаваме директни пътища между някои върхове и представителя.
- Когато пускаме ребро между u и v, компонентата, която е с по-нисък „ранк“, винаги става част от другата компонента.

Амортизирана сложност: $O(n * \alpha(n))$, където е обратната функция на Акерман(която расте много бавно и за всяко разумно n не надвишава 4).

DSU с undo

Преди да решим пълната задача, ще я решим в по-лесния й вариант. Ще трием последното добавено ребро, което все още не е махнато от графа. С други думи ще правим **undo**.

Как правим **undo**? – най-общо казано, ще искаме да върнем състоянието на масивите ни в предишното състояние. Ще използваме стек, в който ще пазим какви промени правим след всяко добавено ребро. Ще пазим и в какъв момент от време правим тези промени. Промените, направени при добавянето на едно ребро, ще бъдат добавени добавени в стека с един и същ момент.

Стекът ни ще изглежда по този начин:

```
struct Change {
        int timeD; //момент на добавяне на реброто
        int* elem; //пойнтер към клетката, която променяме
        int last_val; //стойност на клетката преди да направим промяната
};
stack <Change> st;
```

Да кажем, че в кода имаме следната операция:

```
par[u]=v; //u става част от компонентата на v
```

преди тази операция, ще добавим следния ред в кода:

```
st.push({currTime,par+u,par[u]});
```

Ето примерна имплементация на функцията union:

```
void union(int u, int v) {
    currTime++;
    u=root(u); v=root(v);
    if (u==v) return;
    if (sz[u]>sz[v]) swap(u,v); //v ще бъде родител на u
    st.push({currTime,par+u,par[u]});
    st.push({currTime,sz+v,sz[v]});
    par[u]=v;
    sz[v]+=sz[u];
}
```

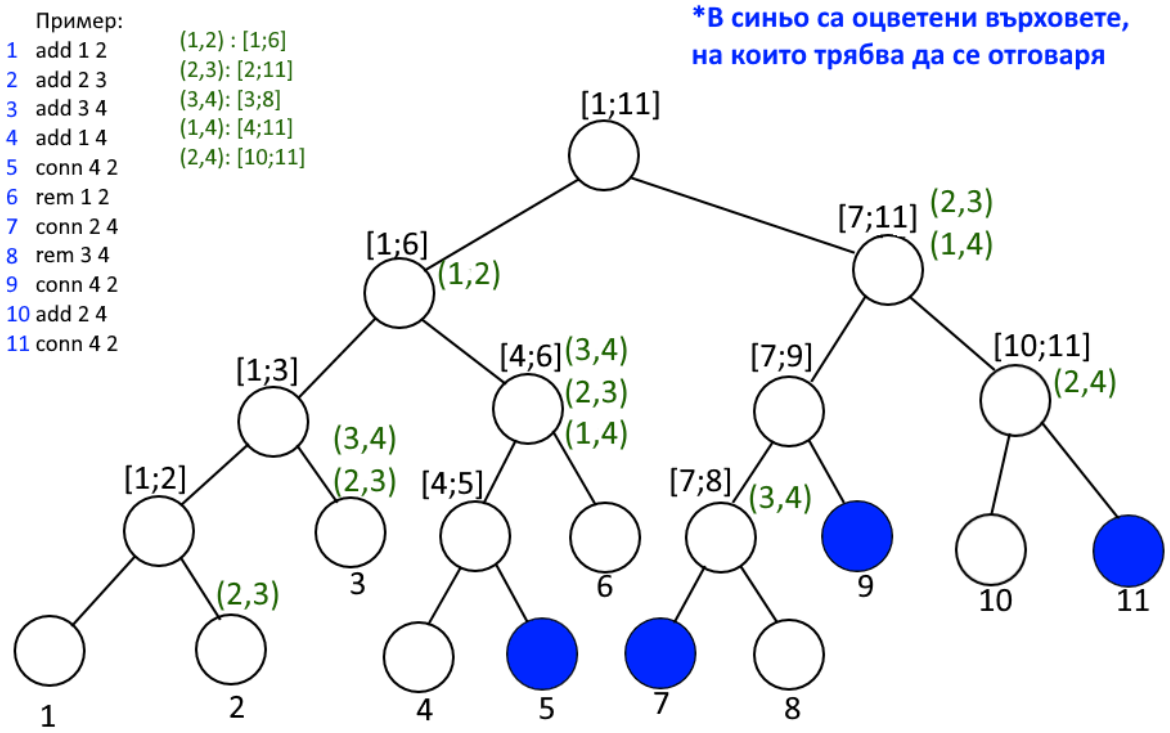
Единственото, което остава, е да напишем функцията **undo**:

```
void undo() {
    while (!st.empty() && st.top().timeD==currTime) {
        auto curr=st.top();
        st.pop();
        (*curr.elem)=curr.last_val;
    }
    currTime--;
}
```

Остава да уточним каква е сложността на алгоритъма. Ако ползваме двете оптимизации на DSU, сложността ни няма да е $O(Q * \alpha(n))$, а $O(Q \log n)$. Защо се случва това?

Сложността на DSU-то с двете оптимизации е амортизирана. Тоест за някоя заявка можем да правим много операции, но сумарно всички операции от всички заявки ще са малко. Когато добавим **undo** в задачата, можем да нагласим заявките така, че да добавяме и изтриваме заявка, която прави много операции. Изводът е да внимаваме с **undo**, когато имаме амортизирана сложност.

Амортизираната сложност на DSU идва от компресията на пътя до корена. Какво става, когато я махнем? Сложността, ще се запази $O(Q \log n)$, тъй като оптимизацията по размер балансира дървото. Следователно, когато викаме root(int v), ще правим най-много $\log n$ операции. Разликата е, че няма да правим ненужни операции в **undo** функцията и ще намалим значително константата на алгоритъма.



Ред на добавяне и махане в DSU-то: +(1,2) +(2;3) undo(2,3) +(3,4) +(2,3) undo(2,3) undo(3,4) +(3,4) +(2,3) +(1,4) undo(1,4) undo(2,3) undo(3,4) +(2,3) +(1,4) +(3,4) undo(3,4) +(2,4)

Решение на пълната задача

За всяко ребро ще пазим в какъв интервал от заявки е „активен“(присъства в графа). В сегментно дърво ще пазим вектор на всеки връх, където ще добавяме ребра, които са активни в интервалът, определен от този връх. За всеки интервал на активност ще добавяме реброто, подобно на range update, в $\log n$ върха на сегментното.

След като сме добавили всички интервали на активност, остава да отговорим на заявките. Ще направим пълно обхождане на сегментното дърво с DFS. Когато стъпим на някой връх, ще добавяме в DSU-то всички ребра, които отговарят за интервалът, определен от този връх(тези ребра вече сме ги добавили с range update). Когато push-ваме в стека промените, случили се в този връх, ще ги обозначаваме с едно и също време. Когато стигнем някое листо, проверяваме дали имаме заявка, на която трябва да отговорим. Ако има, правим проста проверка дали двата върха, за които ни питат, имат един и същи корен. Когато сме приключили обхождането за даден връх в сегментното, е време да се върнем нагоре. Тук идва частта с **undo**. Искаме да върнем DSU-то в състоянието преди обхождането на дадения връх. Това лесно можем да го постигнем с нашата **undo** функция, която ще върне промените до предишния момент(преди да сме влезли в съответния връх в сегментното).

Задачи и материали

- USACO Guide
- cp-algorithms
- Задача DSU с undo
- Основната задача на лекцията
- Имплементация на DSU с триене
- Codeforces Educational round 62 Problem F
- RMI 2018 colors
- Летен 2021 A2.renovation

Автор: Велислав Гърков