Sistem disjunktnih skupova

Razmotrićemo strukturu sistem disjunktnih skupova (engl. "disjoint-set-union", ruski "**"система непересекающихся множеств"**, skraćeno DSU).

Na početku imamo nekoliko elemenata i svaki se nalazi u svom skupu. Operacije su: spojiti neka dva skupa (tj, napraviti njihovu uniju) i ispitati u kom se skupu trenutno nalazi neki element. U klasičnoj varijanti, postoji još i operacija dodavanja novog elementa koji se smješta u odvojeni skup. Osnovne funkcije su:

- `make_set(x)` – dodaje novi element `x`, smješta ga u novi skup koji ima samo taj jedan element
- `union_sets(x,y)` – unija dva skupa (skupa koji sadrži `x` i skupa koji sadrži `y`)
- `find_set(x)` – vraća u kojem je skupu element `x`. U stvari, vraćamo jedan element tog skupa koji nazivamo predstavnikom ili liderom skupa (engl. `leader`). Lider se bira u svakom skupu u strukturi podataka i može se mijenjati (naime poslije poziva `union_sets()`)

Na primjer, ako poziv `find_set()` za neka dva elementa vraća istu vrijednost, to znači da ti elementi pripadaju istom skupu, a inače su u različitim skupovima. Naša struktura je takva da se svaka od navedenih operacija obavlja u prosjeku za `O(1)`.

Efikasna struktura podataka

Skupove elemenata čuvaćemo u obliku drveta: jedno drvo odgovara jednom skupu. Korijen drveta biće lider tog skupa. Uvodimo niz `parent`, u kojem za svaki element čuvamo pokazivača na njegovog pretka u drvetu. Za korijen drveta, smatraćemo da je njihov predak sam korijen.

Naivna realizacija

Sada možemo napisati prvu veziju strukture. Ona neće biti efikasna, pa ćemo je kasnije poboljšati uvođenjem dvije heuristike, tako da dobijemo skoro konstantno vrijeme izvršavanja operacija. Sva informacija se čuva u nizu `parent`. Da bi dodali novi element (operacija `make_set(v)`), samo kreiramo drvo sa korijenom u `v` i postavimo da je njegov predak on sam. Da bi spojili dva skupa (operacija `union_sets(a,b)`) prvo nađemo lidere skupova u kojim se nalaze `a` i `b`. Ako su lideri isti, ništa ne radimo – to znači da skupovi već spojeni. Ako lideri nisu jednaki, onda prosto možemo reći da je predak `b` jednak `a` (ili obratno), i na taj način smo spojili skupove. Na kraju, operacija traženja lidera (`find_Set(v)`) je jednostavna: krećemo se po precima čvora `v`, sve dok ne dođemo do korijena tj. dok pokazivač ne pokazuje na samog sebe. Zbog optimizacija koje ćemo kasnije uraditi, pogodnije je da ova operacija bude rekurzivna.

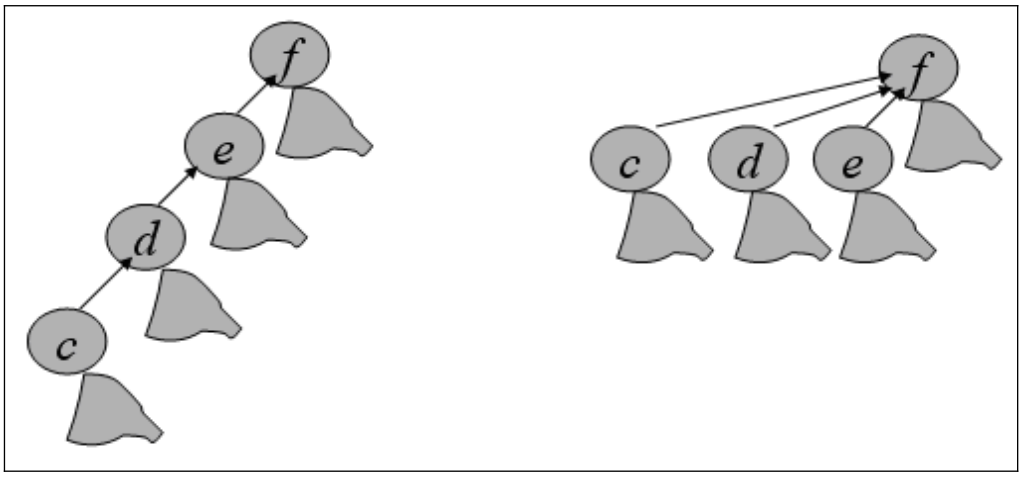
```
void make_set (int v) {
	parent[v] = v;
}

int find_set (int v) {
	if (v == parent[v])
		return v;
```

```
        return find_set (parent[v]);
}

void union_sets (int a, int b) {
        a = find_set (a);
        b = find_set (b);
        if (a != b)
                parent[b] = a;
}
```

Međutim, ova implementacija je daleko od efikasne. Lako je napraviti primjer u kojem će drvo koje predstavlja skup u stvari biti dugačka lista, pa će svaki poziv funkcije `find_set` biti proporcionalan dužni liste tj. `O(n)`. To je mnogo gore od onoga što mi želimo.

Heuristika kompresije puteva

Heuristika kompresije (sažimanja) puteva je predviđena za ubrzanje rada funkcije `find_set()`. Ideja je sljedeća: poslije poziva `find_set(v)`kada nađemo željenog lidera `p` za skup, zapamtimo da je za čvor `v` i sve čvorove po koji su njegovi preci lider upravo `p`. To je najlakše uraditi tako što za sve te čvorove preusmjerimo `parent` da pokazuje upravo na `p`. Na taj način se donekle mijenja smisao niza `parent`: sad je on sažeti niz predaka tj. za svaki čvor se u nizu ne čuva neposredni predak, već predak pretka, predak pretkovog pretka, itd. Sa druge strane, jasno je da ne možemo dopustiti da svi elementi niza `parent` uvijek pokazuju na lidera, jer bi poslije svake operacije `union_sets` morali obnavljati niz `parent` u `O(n)` elemenata. Zbog toga, nizu `parent` pristupamo kao nizu predaka, koji je moguće, djelimično sažet. Na slici je prikazan efekat kompresije puteva.



Nova implementacija ima sljedeći oblik:

```
int find_set (int v) {
        if (v == parent[v])
                return v;
        return parent[v] = find_set (parent[v]);
}
```

Ova implementacija radi sve što nam je trebalo: prvo pomoću rekurzivnih poziva nalazi lidera skupa, a zatim u procesu povratka sa steka poziva tog lidera dodjeljuje svim čvorovima po

kojima je prošla funkcija. Može se napisati i nerekurzivna verzija ove funkcije, ali tada se mora dva puta proći kroz niz: prvi put da se nađe lider, a drugi put da se preusmjere preci za čvorove na putu do lidera. Međutim, u praksi, nerekurzivna verzija ne donosi nikave značajne prednosti u odnosu na rekurzivnu.

Složenost operacija uz heuristiku kompresije puteva

Dokazaćemo da je složenost u prosjeku logaritamska: `O(logn)`. Primjetimo da kako operacija `union_sets` predstavlja dva poziva operacije `find_set()` i još `O(1)` operacija, dovoljno je da odredimo složenost operacije `find_set()`. Uvedimo pojam težine čvora `v` tj. broja potomaka tog čvora, uključujući i samog `v`, i označimo ga sa `w[v]`. Težine čvorova se u procesu rada algoritma mogu samo uvećavati. Uvedimo i pojam raspona grane `(a,b)` kao `|w[a]-w[b]|` (jasno je da je težina pretka veća od težine potomka). Raspon proizvoljne grane može se samo povećavati tokom rada algoritma. Grane podijelimo u klase na sljedeći način: grana ima klasu k ako raspon grane pripada intervalu $\lfloor 2^k, 2^{k+1} - 1 \rfloor$, pa će klasa grane biti cio broj iz intervala $\lfloor 0, \lceil \log n \rceil \rfloor$. Fiksirajno proizvoljan čvor `x` i pogledajno kako se mijenja grana u njegovom pretku: na početku je nema, jer je `x` lider; zatim imamo granu iz `x` u neki drugi čvor (kada se skup koji sadrži `x` spaja sa nekim drugim skupom) i zatim se mijenja pri kompresiji puteva u pozivu find_set, Samo nas interesuje posljednji slučaj, jer prva dva slučaja imaju složenost `O(1)` za jedan upit.

Razmotrimio rad operacije `find_set`: prolazimo kroz drvo duž nekog puta, brišemo sve grane sa tog puta i preusmjeravamo ih u lidera. Posmatrajmo taj put i isključimo iz njega posljednju granu iz svake klase (tj. najviše po jednu granu iz svake od klasa 0,1,..., $\lceil \log n \rceil$). Na taj način smo isključili O($\lceil \log n \rceil$) grana iz svakog upita. Razmotrimo sada sve ostale grane tog puta. Za svaku takvu granu klase `k`, ima još jedna grana klase `k` u putu (inače bi morali isključiti baš tu jednu granu klase `k`). Tada će, poslije kompresije, ta grana biti zamijenjena sa granom klase bar `k+1`. Pošto se raspon grane ne smanjuje, dobijamo da za svaki čvor koji je obišla operacija `find_set`, grana u njegovom pretku je ili isključena ili uvećala klasu. Otuda se dobija složenost za m upita: `O((n+m)logn)`, što za `m>=n` daje logaritamsko vrijeme za jedan upit.

Heuristika spajanja po rangu

Razmotrićemo heuristiku koja sama za sebe ubrzava rad algoritma a u paru sa heuristikom kompresije puteva dostiže praktično konstanto vrijeme rada u prosjeku. Ova heuristika podrazumijeva manju izmjenu operacije `union_set`: spajanje drveta obavljamo po rangu drveta a ne slučajno, kako je to bio slučaj kod naivne implementacije. Postoje dvije vrste heuristike po rangu: u jednoj je rang drveta broj čvorova u njemu a u drugom je dubina drveta (tačnije, gornja granica na dubinu drveta, jer pri kompresiji puteva stvarna dubina drveta se može smanjiti). U obje varijante, suština heuristike je ista: pri izvršenju operacije `union_sets` drvo sa manjim rangom pripajamo drvetu sa većim rangom.

Implementacija gdje je rang **broj čvorova u drvetu**:

```
void make_set (int v) {
	parent[v] = v;
	size[v] = 1;
}
```

```
void union_sets (int a, int b) {
	a = find_set (a);
	b = find_set (b);
	if (a != b) {
		if (size[a] < size[b])
			swap (a, b);
		parent[b] = a;
		size[a] += size[b];
	}
}
```

Implementacija gdje je rang **dubina drveta:**

```
void make_set (int v) {
	parent[v] = v;
	rank[v] = 0;
}

void union_sets (int a, int b) {
	a = find_set (a);
	b = find_set (b);
	if (a != b) {
		if (rank[a] < rank[b])
			swap (a, b);
		parent[b] = a;
		if (rank[a] == rank[b])
			++rank[a];
	}
}
```

Obje varijante sa aspekta složenosti su jednake, pa se u praksi može primjenjivati bilo koja od njih.

Složenost operacija uz heuristiku ranga

Dokazaćemo da je prosječna složenost algoritma koji koristi samo ovu heuristiku (bez kompresije puteva) logaritamska $O(\log n)$ za jedan upit. Ako pokažemo da je dubina drveta $O(\log n)$ za bilo koju varijantu heuristike, to će automatski značiti da logaritamsku složenost za `find_set` a samim tim i za upite tipa `union_sets`.

Neka je rang dubina drveta. Dokažimo da ako je rang drveta k, tada to drvo sadrži bar 2^k čvorova, što znači da je rang $O(\log n)$. Dokaz izvodimo indukcijom. Za $k=0$, tvrđenje očigledno važi. Pri kompresiji puteva, dubina drveta se samo može umanjiti. Rang drveta se uveličava sa $k-1$ na k kada se drvetu prispaja drvo ranga $k-1$. Po induktivnoj pretpostavci, u svakom od drveta ranga $k-1$ ima bar 2^{k-1} čvorova, pa u novom drvetu ima bar 2^k čvorova, što je i trebalo dokazati

Neka je sada rang veličina drveta. Dokažimo da ako je broj čvorova u drvetu k, tada je visina drveta ne veća od $\lfloor \log k \rfloor$. Opet primjenimo indukciju. Za $k=1$, tvrđenje je tačno. Kompresija puteva nam ne smeta, jer se dubina može samo umanjiti. Pokušajmo da spojimo dva drveta veličina k_1 i k_2. Po induktivnoj pretpostavci, dubine tih drveta nisu veće od $\lfloor \log k_1 \rfloor$ i $\lfloor \log k_2 \rfloor$ respektivno. Ne umanjujući opštost, pretpostavimo da je prvo drvo veće ($k_2 \leq k_1$). Tada će

dubina poslije spajanja za k_1+k_2 čvorova biti $h = \max(\lfloor \log k_1 \rfloor, 1 + \lfloor \log k_2 \rfloor)$. Sada treba dokazati da važi: $h \leq \lfloor \log(k_1 + k_2) \rfloor$, odnosno logaritmovanjem obje strane $2^h = \max(2^{\lfloor \log k_1 \rfloor}, 2^{\lfloor \log k_2 \rfloor}) \leq 2^{\lfloor \log(k_1+k_2) \rfloor}$
Ovo je skoro očigledno, jer je $k_1 \leq k_1+k_2$ i $2k_2 \leq k_1+k_2$.

Primjena obje heuristike

Primjena obje heuristike daje najbolje rezultate. Dokaz složenosti je izuzetno dug i matematički zahtijevan, pa ga ne dajemo. Koga zanima, može naći dokaz u knjizi „Introduction To Algorithms“, autori Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest.

Konačni rezultat je takav: vrijeme obrade jedne operacije je je u prosjeku $O(\alpha(n))$, gdje $\alpha(n)$ označava inverznu Akermanovu funkciju, koja raste izuzetno sporo pa se smatra skoro konstantom (za vrijednosti $n<10^{600}$, $\alpha(n)$ ne prelazi 4). Upravo zbog toga se za DSU govori „skoro konstantno vrijeme rada“.

Implementacija koja kombinuje kompresiju puteva i heuristiku ranga, gdje je rang dubina drveta:

```
void make_set (int v) {
	parent[v] = v;
	rank[v] = 0;
}

int find_set (int v) {
	if (v == parent[v])
		return v;
	return parent[v] = find_set (parent[v]);
}

void union_sets (int a, int b) {
	a = find_set (a);
	b = find_set (b);
	if (a != b) {
		if (rank[a] < rank[b])
			swap (a, b);
		parent[b] = a;
		if (rank[a] == rank[b])
			++rank[a];
	}
}
```

Primjena DSU u zadacima i neka poboljšanja

Razmotrićemo neke primjere upotrebe DSU, kako trivijalne tako i one koji zahtijevaju poboljšanje same strukture.

Podrška za komponete povezanosti grafa

Ovo je jedna od očiglednih primjena DSU, koja je, po svemu sudeći, i inicirala izučavanje ove strukture.

Formalno, zadatak je sljedeći: dat je prazan graf i postepeno mu se dodaju grane (prvi tip upita); drugi tip upita je `(a,b)` – da li su `a` i `b` u istoj komponenti povezanosti? Direktna primjena gore opisanih struktura daje nam rješenje koje u prosjeku obrađuje bilo koji upit za konstantno vrijeme. Uzimajući u obzir da se praktično isti takav zadatak pojavljuje i kod Kruskalovog algoritma (vidi poglavlje „Kruskalov algoritam“), odmah dobijamo poboljšanu verziju algoritma koja praktično radi za linearno vrijeme.

Ponekad se u praksi pojavljuje obrnuta verzija ovog zadatka: na početku je dat graf sa nekim čvorovima i granama, i imamo upite tipa „obrisati granu `(a,b)`“. Ako je zadatak off-line, tj. svi upiti su poznati unaprijed, tada zadatak možemo riješiti na sljedeći način: okrenemo zadatak naglavačke tj. smatramo da imamo prazan graf u koji dodajemo grane – prvo posljednju granu iz upita, pa pretposljednju, itd. Tako smo dobili naš obični zadatak, koji smo ranije riješili.

Komponente povezanosti na slici

Data je slika sa `nxm` piksela, gdje su svi pikseli na početku bijeli. Zatim su neki pikseli obojani crnom bojom. Odrediti dimenzije svake bijele komponente povezanosti na konačnoj slici.

Opet koristimo DSU na sljedeći način: posmatramo sve bijele piksele i njihova četiri susjeda; ako je susjed takođe bijeli piksel, pozivamo `union_sets` za ta dva piksela. Na početku imamo `nxm` čvorova, koji odgovaraju pikselima slike. Drveta koja dobijemo poslije završetka algoritma predstavljaju tražene komponente povezanosti.

Podrška dopunskoj informaciji za skupove

DSU dozvoljava čuvanje bilo kakve dopunske informacije koja se odnosi na skupove. Prosti primjer je čuvanje dimenzije skupa, kako je opisano u implemenatciji ranga po heuristici (tamo je to opisamo za lidere skupova). Zajedno sa liderom možemo čuvati svaku informaciju koja nam je potrebna za konkretan zadatak. Međutim, takvo rješenje primjenom DSU nema nikavih prednosti u odnosu na rješenje primjenom obilaska u dubinu (DFS).

Primjena DSU za sažimanje „skokova“ po segmentu. Zadatak o bojanju segmenata u off-line režimu

DSU se često primjenjuje u slučajevima kada imamo neku kolekciju elemenata i iz svakog elementa vodi jedna grana. Tada možemo brzo (za skoro konstatno vrijeme) naći krajnju tačku u koju dolazimo ako se krećemo po datoj grani iz date polazne tačke.

Primjer takvog zadatka je bojanje segmenata: dat je segment dužine `L`, gdje je svaki jedinični segment obojan bojom `0`. Zatim imamo upite oblika `(l,r,c)` – segment `[l,r]` obojati bojom `c`. Potrebno je odrediti konačnu boju svakog jediničnog segmenta. Upiti su poznati unaprijed, pa je zadatak off-line.

Uvedimo DSU koja će za svaki jedinični segment čuvati pokazivač na najbliži neobojeni jedinični segment desno od tekućeg segmenta. Na početku, svaki segment pokazuje na samog sebe. Sada posmatrajmo upite u obrnutom poretku, od posljednjeg ka prvom. Za izvršenje upita, mi pomoću DSU nalazimo krajnju lijevi neobojani segment, obojimo ga i preusmjerimo sve pokazivače iz njega na sljedeći prazan segment desno od njega. Faktički, koristimo DSU sa kompresijom puteva ali bez heuristike po rangu, jer nam je važno ko je lider

poslije spajanja. Zbog toga je složenost $O(\log n)$ po upitu, pri čemu je konstanta sakrivena u $O()$ relativno mala u odnosu na ostale metode.

Implementacija:

```
void init() {
	for (int i=0; i<L; ++i)
		make_set (i);
}

void process_query (int l, int r, int c) {
	for (int v=l; ; ) {
		v = find_set (v);
		if (v >= r)  break;
		answer[v] = c;
		parent[v] = v+1;
	}
}
```

Međutim, moguće je koristiti i heuristiku po rangu: za svaki skup u nekom nizu `end[]` čuvamo gdje se taj skup završava (tj. krajnju desnu tačku). Tada je moguće spajati skupove primjenom heuristike po rangu i postavljajući novodobijenom skupu novu desnu granicu. Na taj način dobijamo rješenje složenosti $O(\alpha(n))$.

Podrška rastojanju do lidera

Ponekad se u konkretnim primjenama DSU pojavljuje potreba za rastojanjima do lidera tj. broja grana od korijena drveta do tekućeg čvora. Ako ne bi bilo kompresije puteva, tada bi se to jednostavno realizovalo – to bi bio broj rekurzivnih poziva u funkciji `find_set`. Kod kompresije puteva, više grana mogu biti sažete u jednu. Zbog toga je potrebno za svaki čvor čuvati i dodatnu informaciju – dužinu tekuće grane iz čvora do pretka. U implementaciji je zgodno da niz `parent[]` i funkcija `find_set` umjesto jedne vrijednosti vraćaju par vrijednosti (lider, rastojanje do lidera):

```
void make_set (int v) {
	parent[v] = make_pair (v, 0);
	rank[v] = 0;
}

pair<int,int> find_set (int v) {
	if (v != parent[v].first) {
		int len = parent[v].second;
		parent[v] = find_set (parent[v].first);
		parent[v].second += len;
	}
	return parent[v];
}

void union_sets (int a, int b) {
	a = find_set (a) .first;
	b = find_set (b) .first;
	if (a != b) {
```

```
                if (rank[a] < rank[b])
                        swap (a, b);
                parent[b] = make_pair (a, 1);
                if (rank[a] == rank[b])
                        ++rank[a];
        }
}
```

Podrška parnosti dužine puta i zadatak o provjeri bipartitnosti grafa u online režimu

Po analogiji sa dužinom puta do lidera, moguće je podržavati i parnost puta do lidera. Zašto smo ovaj problem razdvojili od prethodnog?

Potreba za parnošću puta pojavjuje se u sljedećem zadatku: dat je prazan graf, u koji se dodaju grane a pojavljuju se i upiti tipa „da li je komponenta koja sadrži dati čvor bipartitni graf?“

Možemo uvesti DSU za komponente povezanosti i u svakom čvoru čuvamo parnost puta do njenog lidera. Lako možemo provjeriti da li dodavanje konkretne grane narušava bipartitnost grafa ili ne. Ako krajevi grane pripadaju istoj komponenti povezanosti i imaju istu parnost puta do lidera, tada dodavanje te grane kreira ciklus neparne dužine i komponenta više nije bipartitni graf. Glavna smetnja sa kojom se susrećemo je kako da spojimo dva drveta uzimajući u obzir parnost puteva. Ako dodamo granu `(a,b)` koja spaja dvije kompoonente povezanosti u jednu, tada pri pripajanju jednog drveta drugom moramo mu dodijeliti parnost tako da bi a i b imali različite parnosti. Izvedimo formulu koja nam to omogućava. Neka je `x` parnost čvora `a`, `y` – parnost čvora `b` i `t` – tražena parnost koju treba dodijliti lideru. Ako skup sa čvoron a pripojimo skupu sa čvorom `b`, tada se `y` ne mijenja, dok u čvoru `a` mijenjamo parnost na `x^t`, gdje je `^` operator „bit po bit ekskluzivno ili“ (engl. bitwise xor). Nama odgovara da se te parnosti razlikuju, tj. dobijamo jednačinu `x^t^y=1`, čije je rješenje `t=x^y^1`.

Kao i u prethodnom poglavlju, koristićemo parove. Za svaki skup (komponentu povezanosti), u nizu `bipartite[]` čuvamo informaciju da li je bipartitni ili ne.

```
void make_set (int v) {
        parent[v] = make_pair (v, 0);
        rank[v] = 0;
        bipartite[v] = true;
}

pair<int,int> find_set (int v) {
        if (v != parent[v].first) {
                int parity = parent[v].second;
                parent[v] = find_set (parent[v].first);
                parent[v].second ^= parity;
        }
        return parent[v];
}
```

```
void add_edge (int a, int b) {
	pair<int,int> pa = find_set (a);
	a = pa.first;
	int x = pa.second;

	pair<int,int> pb = find_set (b);
	b = pb.first;
	int y = pb.second;

	if (a == b) {
		if (x == y)
			bipartite[a] = false;
	}
	else {
		if (rank[a] < rank[b])
			swap (a, b);
		parent[b] = make_pair (a, x ^ y ^ 1);
		if (rank[a] == rank[b])
			++rank[a];
	}
}

bool is_bipartite (int v) {
	return bipartite[ find_set(v) .first ];
}
```

RMQ (minimum na segmentu) za O(α(n)) u prosjeku u off-line verziji

Formalno, zadatak je sljedeći: realizovati strukturu podataka koja podržava dva oblika upita: dodavanje broja `insert (i), i =1,...,n` i traženje i uklanjanje tekućeg minimalnog broja `extract_min()`. Smatraćemo da se svaki broj dodaje tačno jednom i da su nam upiti poznati unaprijed (tj. zadatak je off-line).

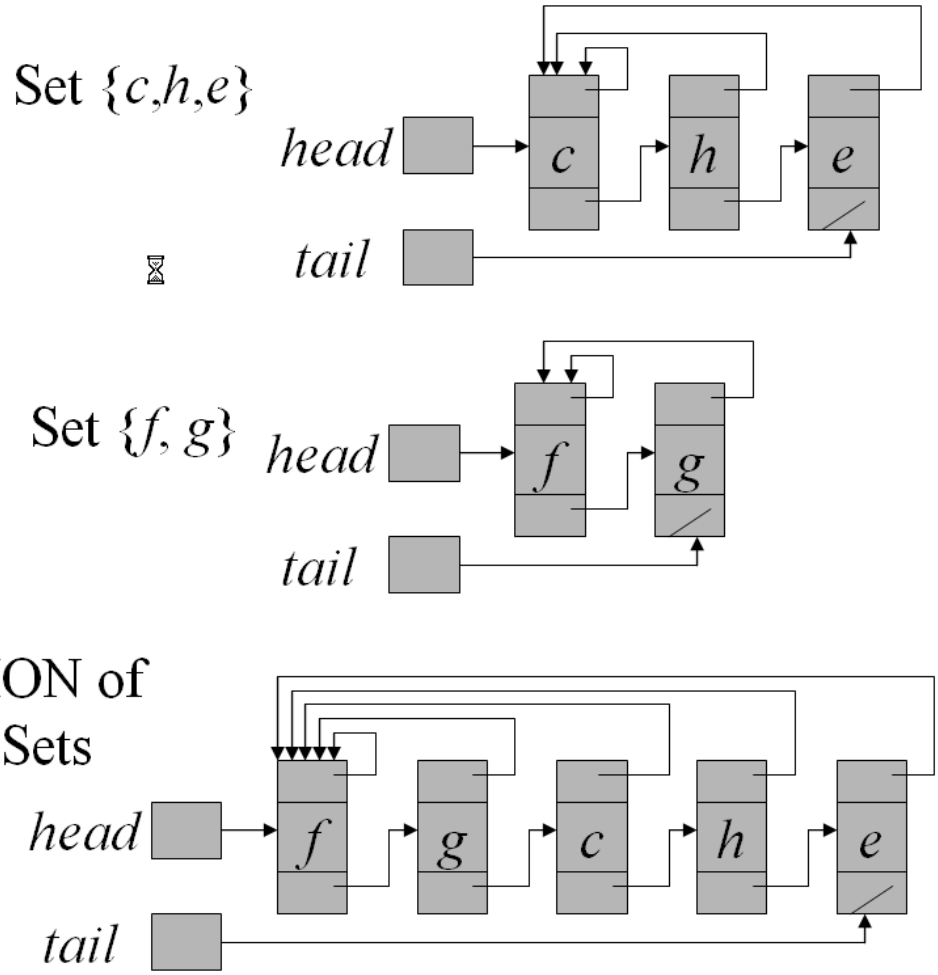
Ideja rješenja je sljedeća: umjesto da po redu odgovaramo na upite, prođimo kroz brojeve `i=1..n` i odredimo koji je to upit čiji je odgovor upravo broj `i`. Za to je potrebno naći prvi neodgovoreni upit koji ide poslije upita `insert(i)` (koji je dodao broj u strukturu) – lako se vidi da je upravo to upit čiji je odgovor i. Na taj način, imamo situaciju sličnu zadatku o bojanju segmenata. Možemo dobiti rješenje za `O(logn)` u prosjeku po upitu ako ne koristimo heuristiku po rangu, u svakom elementu čuvamo pokazivač na najbliži upit tipa `extract_min` desno od tekućeg elementa i koristimo kompresiju puteva. Postoji i rješenje za `O(α(n))` ako koristimo heuristiku po rangu i u svakom skupu čuvamo broj pozicije gdje se skup završava. U prethodnoj varijanti, ta se informacija automatski pojavljivala jer su pokazivači išli udesno – sad ih treba čuvati eksplicitno

LCA (najmanji zajednički predak u drvetu) za O(α(n)) u prosjeku u off-line verziji

Algortam Tarjana za nalaženje LCA za `O(1)` u prosjeku u online režimu opisan je u posebnom poglavlju. Razlikuje se od drugih algoritama LCA svojom jednostavnošću, posebno u poređenju sa optimalnim algoritmom Farah-Koloton-Bendera.

Čuvanje DSU u obliku eksplicitne liste skupova

Alternativni način čuvanja DSU je da svaki skup predstavljamo pomoću ulančane liste njegovih elemenata. U svakom elementu čuva se i pokazivač na lidera skupa. Na prvi pogled izgleda da to nije efikasno: pri spajanju dva skupa moramo jednu listu nadovezati na drugu i također obnoviti lidera kod svih elemenata jedne od listi. Ipak, pokazuje se da težinska heuristika (tj. pravilo da manji skup dodajemo većem) dozvoljava suštinski sniziti složenost do `O(m+nlogn)` za `m` upita nad `n` elemenata. Pripajanje jednog skupa drugom moguće je uraditi za vrijeme proporcionalno broju elemenata u manjem a funkciju `find_set` za vrijeme `O(1)`.



Dokažimo složenost `O(m+nlogn)` za `m` upita. Fiksirajmo element `x` i pogledajmo kako na njega utiče operacija `union_sets`. Kada se to dogodi prvi put, skupa u kojem je `x` ima bar `2` elementa. Kada se to dogodi drugi put, skup će imati bar `4` elementa (jer u veći skup dodajemo manji), itd. Dobijamo da nad `x` možemo uraditi najviše $\lceil \log n \rceil$ spajanja, Na taj način je, po svim čvorovima `O(nlogn)` i još `O(1)` za svaki upit, što ukupno daje O(m+nlogn) .

Primjer implementacije:

```
vector<int> lst[MAXN];
int parent[MAXN];
```

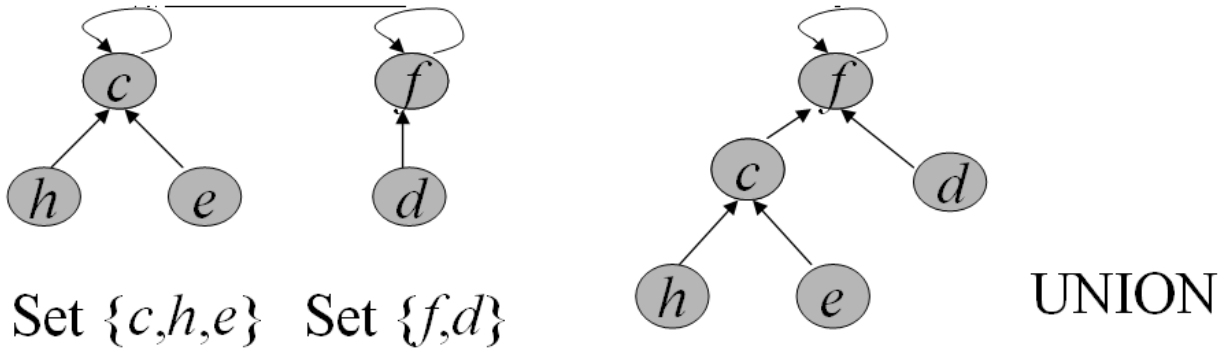
```
void make_set (int v) {
	lst[v] = vector<int> (1, v);
	parent[v] = v;
}

int find_set (int v) {
	return parent[v];
}

void union_sets (int a, int b) {
	a = find_set (a);
	b = find_set (b);
	if (a != b) {
		if (lst[a].size() < lst[b].size())
			swap (a, b);
		while (!lst[b].empty()) {
			int v = lst[b].back();
			lst[b].pop_back();
			parent[v] = a;
			lst[a].push_back (v);
		}
	}
}
```

Ideju pripajanja manjeg skupa većem možemo iskoristiti i u zadacima gdje se ne koristi DSU. Na primjer, dat je sljedeći zadatak: dato je drvo tako da je u svakom čvoru tog drveta upisan neki broj, pri čemu se jedan te isti broj može pojavljivati na više mjesta. Za svaki čvor drveta treba naći broj različitih brojeva u njegovom poddrvetu. Ako primjenimo istu ideju, možemo dobiti sljedeće rješenje: pokrenemo DFS po drvetu, koji će vraćati pokazivač na strukturu `set` – listu svih brojeva u poddrvetu. Da bi dobili odgovor za tekući čvor (ako on nije list), potrebno je pokrenuti DFS za svu djecu tog čvora i spojiti sve dobijene strukture `set` u jednu. Broj elemenata u toj strukturi `set` biće traženi odgovor. Spajanje više skupova u jedan obavljamo po principu „manji skup dodajemo u veći“. Konačno dobijamo rješenje složenosti $O(n \log^2 n)$, jer dodavanje jednog elementa u `set` ima složenost $O(\log n)$.

Čuvanje DSU u obliku eksplicitne strukture drveta

Jedna od moćnih primjena DSU je u tome što istovremeno može čuvati i sažetu (komprimovanu) i nesažetu strukturu drveta. Sažeta struktura se koristi brzo spajanje drveta i provjeru pripadnosti čvorova drvetu, a nesažeta npr. za traženje puta između dva čvora ili drugih obilazaka strukture drveta.

Kod implementacije, to znači da pored običnog niza sažetih predaka `parent[]`, uvodimo i niz `real_parent[]`, Jasno je da ovaj niz nikako ne utiče na složenost: promjene u njemu se dešavaju samo pri spajanju dva skupa i samo u jednom elementu.



Sa druge strane, u praktičnim primjenama, često treba spojiti dva drveta granom koj ne mora polaziti iz korijena. To znači da nemamo drugog izlaza osim da jedno od drveta moramo „**prelomiti”**, da bi to drvo mogli prispojiti drugom drvetu i korijen tog drveta postavljamo kao sina ka drugom kraju dodate grane. Na prvi pogled izgleda da je ova operacija veoma spora i jako će uticati na složenost. Zaista, da bi prelomili drvo u čvoru `v`, moramo proći od tog čvora do korijena drveta, obnavljajući pokazivače `parent[]` i `real_parent[]`. Međutim, u prosjeku to nije strašno, jer uzimamo manje drvo i složenost jednog spajanja je O(logn). Detalje možete naći u poglavlju „Nalaženje mostova grafa“.

Istorijske napomene

Struktura DSU je poznata odavno. Način memorisanja u obliku šume, je po svemu sudeći, prvi put opisan 1964. godine (Galler, Fisher "An Improved Equivalence Algorithm"), ako je potpuna analiza složenosti prikazana mnogo kasnije.

Heuristike kompresije puta i spajanja po rangu su razradili McIlroy i Morris, i nezavisno do njih Tritter.

Jedno vrijeme je bila samo poznata ocjena O(log*n) u prosjeku, koju su 1973.godine prikazali Hopcroft i Ullman u radu "Set-merging algomthms", gdje log*n označava tzv. iterirani algoritam (to je sporo rastuća funkcija, ali ne toliko sporo kao inverzna funkcija Akermana).

Prvi se put ocjena O(α(n)) pojavila u radu Tarjana iz 1975. godine (Tarjan "Efficiency of a Good But Not Linear Set Union Algorithm"). Kasnije je 1985. godine dobio istu ocjenu za nekoliko različitih heuristika ranga (Tarjan, Leeuwen "Worst-Case Analysis of Set Union Algorithms").

Na kraju, Fredman i Saks su u radu "The cell probe complexity of dynamic data structures" iz 1989. godine dokazali da svaki algoritam DSU dužan raditi za minimum O(α(n)) u prosjeku.

Međutim, postoje radovi koji osporavaju tu ocjenu i tvrde da DSU sa heuristikama kompresije puta i spajanja po rangu, npr.: Zhang "The Union-Find Problem Is Linear", Wu, Otoo "A Simpler Proof of the Average Case Complexity of Union-Find with Path Compression".

Online judges zadaci

Spisak zadataka koji se mogu riješiti primjenom DSU:

- TIMUS #1671 **"Паутина Ананси"** [složenost: niska]
- CODEFORCES 25D **"Дороги не только в Берляндии"** [složenost: srednja]
- TIMUS #1003 **"Чётность"** [složenost: srednja]
- SPOJ #1442 **"Chain"** [složenost: srednja]

Literatura

- Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest - Introduction To Algorithms
- Kurt Mehlhorn, Peter Sanders. **Algorithms and Data Structures: The Basic Toolbox** [2008]
- Robert Endre Tarjan. **Efficiency of a Good But Not Linear Set Union Algorithm** [1975]
- Robert Endre Tarjan, Jan van Leeuwen. **Worst-Case Analysis of Set Union Algorithms** [1985]