Opisi algoritama

Zadatke, testne primjere i rješenja pripremili: Adrian Beker, Marin Kišić, Daniel Paleka, Ivan Paljak, Tonko Sabolčec, Bojan Štetić i Paula Vidas. Primjeri implementiranih rješenja su dani u priloženim izvornim kodovima.

Zadatak: Paint

Pripremili: Tonko Sabolčec i Marin Kišić
Potrebno znanje: teorija grafova, *union find* struktura podataka, metoda spajanja *manji* na *veći*

Prvi podzadatak može se riješiti šetnjom po istobojnom susjedstvu koristeći DFS ili BFS algoritam. Složenost jedne operacije bojenja u tom je slučaju linearno ovisna o broju piksela u komponenti (istobojnom susjedstvu). U najgorem slučaju komponenta će imati $O(R \cdot S)$ piksela, pa je ukupna složenost takvog pristupa $O(R \cdot S \cdot Q)$.

Drugi podzadatak specijalni je slučaj kada se slika sastoji od samo jedne dimenzije. Neke od ideja kako pristupiti tom problemu su:

- Održavanje skupa komponenti (kontinuirani intervali – npr. korištenjem `std::set` u C++) koji omogućava brzo dohvaćanje susjeda i spajanje intervala (brisanje i dodavanje u skup).
- Održavanje povezanih komponenti pomoću *union find* strukture podataka.
- Podjela niza na dijelove veličine $O(\sqrt{S})$ uz održavanje zastavice za svaki dio koja označava čini li taj dio jednu komponentu (tj. je li cijeli dio jednobojan?).

Treći podzadatak ima sljedeću posebnost: u pojedinom koraku ili neće biti nikakvih promjena (ako boja kojom je kantica napunjena odgovara boji komponente koju bojimo) ili ćemo komponenti koju bojimo promijeniti boju i povezati je sa *svim* njezinim susjednim komponentama. Za održavanje komponenti može se koristiti *union find* struktura podataka, uz koju ćemo za svaku komponentu održavati i skup (npr. `std::set` u C++) aktivnih susjeda. Prilikom prolaska po skupu susjeda trenutne komponente, tj. prilikom spajanja trenutne sa susjednom komponentom, trenutnoj komponenti brišemo susjednu komponentu iz skupa (i obrnuto). Spajanja neće biti više od $O(R \cdot S)$, a budući da svako spajanje ima složenost operacije brisanja iz skupa, $O(\log(R \cdot S))$, ukupna je složenost $O(R \cdot S \cdot \log(R \cdot S))$.

Na prvi pogled može djelovati da bi se algoritam za treći podzadatak mogao koristiti i za konačno rješenje. No, problem je u tome što ne moramo spojiti sve susjedne komponente u svakom koraku, a ipak moramo prolaziti po svim susjednim komponentama da provjerimo boju. Možda bismo mogli doskočiti tom problemu tako da održavamo susjede grupirane po bojama (npr `std::map` u C++), no prilikom promjene boje na trenutnoj komponenti, opet bismo morali prolaziti po svim susjedima da ih se obavijesti o promjeni boje trenutne komponente. Imajmo zasad na umu spomenute opservacije, no krenimo ipak od rješenja prvog podzadatka. Primijetite da, ako bi prosječna veličina komponenti koje bojimo bila dovoljno mala, primjerice oko 10 piksela, tada bi algoritam za prvi podzadatak bio dovoljno brz. Naime, ukupna složenost iznosila bi $O(R \cdot S + Q \cdot 10)$. No, što ako je prosječna veličina komponenti veća? Za prosječnu veličinu komponenti 100 piksela složenost je i dalje dovoljno dobra. Za 1000 piksela vjerojatno također. Za 10000 piksela? Teško. Međutim, primijetite da povećavanjem te granice smanjujemo broj komponenti koji tu granicu nadmašuju. Drugim riječima, u svakom trenutku postoji najviše $R \cdot S/K$ komponenti koje se sastoje od K ili više piksela (o odabiru konkretne vrijednosti K bit će više riječi kasnije). Ovo svojstvo moglo bi se nekako iskoristiti, intuicija je sljedeća: za komponente koje se sastoje od K ili manje piksela, primjenjujemo neki od naivnih pristupa sličnih onom iz prvog podzadatka, dok za komponente s više od K piksela pristupamo na drukčiji način (koji ne bi bio dovoljno učinkovit kad bismo ga primjenjivali na veći broj manjih komponenti).

Konkretno, za svaku komponentu pamtimo sljedeće vrijednosti: *boju*, *veličinu* i *popis velikih susjednih komponenata* (s više od K piksela – primijetite da njih ima najviše $R \cdot S/K$). Prilikom spajanja komponenti i održavanja spomenutih vrijednosti po običaju koristimo *union find* strukturu podataka. Dodatno,

specijalno za *velike* komponente održavamo i *popis susjeda grupiranih po boji*. Algoritam ćemo objasniti pomoću sljedećih tri operacija:

1. Operacija $Oboji(x, y, boja)$ – primjena kantice za ispunu na pikselu (x, y):
   - Određuje se oznaka komponente c kojoj pripada piksel (x, y).
   - Pristupa se operaciji $SusjediUBoji(c, boja)$ za određivanje oznaka susjednih komponenti iste boje, $c'_1, c'_2, ..., c'_l$.
   - Dobivene komponente c'_i spajaju se s trenutnom komponentom c ($c = Spoji(c, c'_i)$).
   - Nakon spajanja, prolazi se po svim velikim susjedima dobivene komponente i *obavještava* ih se o promjeni boje (tj. dodaje se oznaka trenutne komponente c u popis susjedne komponente za zadanu boju).
2. Operacija $SusjediUBoji(c, boja)$ – pronalazak susjednih komponenti zadane boje:
   - Ako je komponenta c mala, prolazi se po svim susjedima komponente (kojih ima $O(K)$) i za svakog provjeri koje je boje.
   - Ako je komponenta c velika, dostupan je popis susjeda po bojama, koji se u tom slučaju izravno koristi.
     Zamjedba: Nakon prolaska po popisu obojenih susjeda, taj se popis može (mora) obrisati (jer nakon spajanja više neće postojati susjed zadane boje – sve će biti dio jedne komponente).
3. Operacija $Spoji(c_1, c_2)$ - spajanje komponenti c_1 i c_2:
   - Veličina nastale komponente dobiva se zbrajanjem veličina komponenti c_1 i c_2.
   - Udruženi popis velikih susjeda dobiva se metodom spajanja manjeg skupa na veći skup. (Moguće je da nakon ove operacije u skupu imamo oznake komponenti koje su u međuvremenu nestale, tj. spojene u neku veću komponentu. Za pristup tom problemu vidi implementacijski detalj niže u tekstu.)
   - U slučaju spajanja dviju velikih komponenti, potrebno je stvoriti i udruženi popis susjednih komponenti po bojama. Pritom se također koristi metoda spajanja manjeg skupa na veći.

Operaciju *Spoji* obavit ćemo najviše $O(R \cdot S)$ puta. Metoda spajanja manje liste na veću osigurava ukupnu amortiziranu složenost $O(R \cdot S \cdot \log(R \cdot S))$. Složenost svih operacija *SusjediUBoji* u slučaju malih komponenti je $O(R \cdot S \cdot K)$, dok je amortizirana složenost u slučaju velikih komponenti $O(R \cdot S)$. Konačno, složenost svih operacija *Oboji* je $O(Q \cdot R \cdot S/K)$. Ukupna složenost ovisi o parametru K i iznosi $O(R \cdot S \cdot \log(R \cdot S) + R \cdot S \cdot K + \frac{Q \cdot R \cdot S}{K})$. Slijedi da se optimalna složenost dobiva za $K = O(\sqrt{Q})$, koji bi za konkretna ograničenja u zadatku iznosio između 100 i 500.

Spomenimo i još jedan **implementacijski detalj**: Prilikom spajanja susjeda, moguće je da će neke oznake u listi velikih susjeda i listi susjeda grupiranih po bojama postati nevažeće ili da će se boja susjeda u međuvremenu promijeniti. Nije potrebno osmišljavati metode kojima će se takve nevažeće komponente pronaći i izbrisati iz popisa. Dovoljno je prilikom prolaska kroz popis dodatno provjeriti trenutno stanje i usporediti ga s očekivanim, preskakujući oznake koje ne zadovoljavaju očekivane uvjete.

Zadatak: Pastiri

Pripremili: Adrian Beker, Paula Vidas, Daniel Paleka
Potrebno znanje: grafovi, pretraga u širinu/dubinu (BFS/DFS)

Neka je V skup čvorova stabla. U opisu rješenja poistovjećivat ćemo ovcu/pastira i čvor u kojem se nalazi. Primijetimo najprije da zadatak možemo promatrati kao instancu takozvanog *set cover* problema. Zaista, ako svakom čvoru $v \in V$ pridružimo skup S_v ovaca koje čuva pastir u v, tada je potrebno naći najmanji podskup $P \subseteq V$ takav da je $\bigcup_{v \in P} S_v$ cijeli skup ovaca. Iako je generalna verzija ovog problema NP-potpuna, specifična struktura dotičnog slučaja omogućit će nam da ga riješimo u odgovarajućoj složenosti.

Krenimo od prvog podzadatka, odnosno slučaja kada je zadano stablo lanac. Tada pastir može čuvati samo prvu ovcu lijevo/desno od sebe, stoga se familija $\{S_v \mid v \in V\}$ sastoji od sljedećih skupova:

- $\{x\}$ za svaku ovcu x;
- $\{x, y\}$ za susjedne ovce x, y takve da su x, y iste parnosti.

Nameće se sljedeći pohlepni algoritam – pogledamo prvu ovcu, u slučaju da je druga ovca iste parnosti, postavimo pastira na pola puta između njih, u suprotnom ga postavimo u čvor prve ovce. Nakon toga uklonimo pokrivene ovce te ponavljamo algoritam. Opisano rješenje ima složenost $\mathcal{O}(N + K)$.

U drugom podzadatku, podskupove ovaca predstavljat ćemo bitmaskama, odnosno brojevima iz skupa $\{0, 1, \ldots, 2^K - 1\}$. Na početku pustimo BFS/DFS iz svake ovce te na taj način u složenosti $\mathcal{O}(K \cdot N)$ odredimo skup S_v za svaki čvor v. Dalje zadatak rješavamo dinamičkim programiranjem. Za svaku bitmasku $mask$ neka $f(mask)$ označava minimalan broj skupova S_v čija unija sadrži $mask$. Iteriramo kroz stanja $mask$ u rastućem poretku. U prijelazu iteriramo kroz sve podmaske $submask$ te ukoliko $submask$ odgovara nekom od skupova S_v, osvježavamo $f(mask)$ vrijednošću $f(mask \wedge submask) + 1$ (ovdje $\wedge$ označava bitovno isključivo ili). Nakon toga preostaje za sve podmaske $submask$ osvježiti $f(submask)$ vrijednošću $f(mask)$. Memorijska je složenost $\mathcal{O}(N + 2^K)$, a vremenska $\mathcal{O}(K \cdot N + 3^K)$ (dokaz ove standardne činjenice ostavljamo čitateljici za vježbu).

Za preostale podzadatke, ukorijenimo stablo u proizvoljnom čvoru. Za svaku ovcu x, njenim *teritorijem* zvat ćemo skup $\{v \in V \mid x \in S_v\}$. Za dvije ovce x, y reći ćemo da su *prijatelji* ako njihovi teritoriji imaju neprazan presjek. Ideja je pohlepno postavljati pastira koji čuva neku ovcu i sve njene dosad nepokrivene prijatelje. To će nam omogućiti sljedeća tvrdnja:

Tvrdnja 1. Za neku ovcu x, neka je $a(x)$ njen najviši predak koji se nalazi u njenom teritoriju. Tada $S_{a(x)}$ sadrži sve prijatelje od x koji nisu dublji od x.

Dokaz. Neka je ovca y prijatelj od x koji nije dublji od x. Tada možemo pretpostaviti da se y ne nalazi u podstablu od $a(x)$ jer u suprotnom je tvrdnja očita. Uzmimo čvor z koji se nalazi u teritorijima od x i y te neka je čvor w polovište puta od x do y. Tada je

$$d(z, x) = d(z, y) = d(z, w) + d(w, x) = d(z, w) + d(w, y),$$

gdje $d(u, v)$ označava udaljenost čvorova u, v. Također, za bilo koju ovcu t vrijedi

$$d(z, w) + d(w, x) = d(z, x) \leq d(z, t) \leq d(z, w) + d(w, t),$$

odakle slijedi $d(w, x) \leq d(w, t)$, a analogno se pokazuje $d(w, y) \leq d(w, t)$. Dakle, w se nalazi u presjeku teritorija od x i y. Štoviše, budući da y nije dublja od x, w je predak od x, stoga se nalazi na putu od x do $a(x)$. Budući da y nije u podstablu od $a(x)$, vrijedi

$$d(a(x), y) \leq d(w, y) = d(w, x) \leq d(a(x), x),$$

stoga pastir u $a(x)$ čuva y, što je i trebalo dokazati. □

Prema Tvrdnji 1, sljedeći je algoritam ispravan:

- Dok god nisu sve ovce pokrivene ponavljaj:
  - Postavi pastira u $a(x)$, gdje je x najdublja dosad nepokrivena ovca.

Direktna implementacija ovog algoritma ima složenost $\mathcal{O}(N(N+K))$ te je dovoljna za ostvariti sve bodove na trećem podzadatku. Za četvrti podzadatak potrebno je ubrzati opisani algoritam. Za svaki čvor v, neka $dep(v)$, $dist(v)$ redom označavaju njegovu dubinu i udaljenost do najbliže ovce. Na početku možemo izračunati $dist(v)$ pomoću BFS-a iz svih ovaca. Alternativno, možemo zamisliti da smo stablu dodali *dummy* čvor povezan sa svim ovcama i iz njega pustili BFS. Sljedeća opservacija sada nam omogućuje da efikasno odredimo čvor $a(x)$ za svaku ovcu x:

Opservacija 2. Ako je čvor v predak ovce x, tada je $dist(v) \leq dep(x) - dep(v)$, i jednakost vrijedi ako i samo ako je v u teritoriju ovce x.

Prema Obzervaciji 2, $a(x)$ je pozicija prvog pojavljivanja maksimuma od $dist(v) + dep(v)$ po svim čvorovima v na putu od korijena do x. Stoga je $a(x)$ za svaku ovcu x moguće izračunati jednostavnim DFS-om iz korijena. Konačno, preostaje efikasno održavati najdublju dosad nepokrivenu ovcu. Ako sortiramo ovce padajuće po dubini, taj se problem svodi na održavanje pokrivenih ovaca. U tu svrhu za početni BFS promotrimo pripadajući *graf najkraćih putova*. To je usmjeren graf G sa skupom vrhova V te bridovima (u, v) gdje je $\{u, v\}$ brid stabla takav da je $dist(v) = dist(u) + 1$.

Opservacija 3. Za svaki čvor $v \in V$, S_v je skup ovaca x takvih da postoji put od x do v u G.

Kad god postavimo novog pastira, proširimo se DFS-om iz njega unatrag po grafu G te pritom pazimo da obilazimo samo dosad neposjećene čvorove. Prema Obzervaciji 3, ovca je pokrivena ako i samo smo ju posjetili u DFS-u. Budući da svaki čvor posjećujemo najviše jednom, održavanje pokrivenih ovaca ima ukupno linearnu složenost.

Na kraju, ukupna je složenost $\mathcal{O}(N + K \log K)$. Primijetimo još da se faktor $\log K$ pojavljuje isključivo zbog sortiranja ovaca po dubini, što je naravno moguće izvesti i u složenosti $\mathcal{O}(N + K)$ jer dubine ne prelaze N. Stoga postoji rješenje i u linearnoj složenosti. Za implementacijske detalje pogledajte službene kodove.

Zadatak: Semafor

Pripremili: Bojan Štetić, Ivan Paljak, Daniel Paleka
Potrebno znanje: matrično množenje, teorija grafova, brzo potenciranje

Kao prvo, stanje semafora predstavljamo kao bitmasku od 5 ili 10 bitova, ovisno o podzadatku. Bitmasku koja predstavlja ispravno stanje nazovimo *dobrom*.

Dobro poznati *graf hiperkocke* za vrhove ima sve bitmaske nekog broja bitova (u ovom zadatku 10), a bridovima su povezani vrhovi koji se razlikuju na točno jednom mjestu. Neka je H matrica susjedstva hiperkocke.

Tada zadatak formalno glasi: pronađite broj šetnji duljine N u hiperkocki od stanja X do svake od dobrih bitmaski, takvih da šetnja svakih K koraka stane u nekoj dobroj bitmaski.

Tvrdnja 1. Broj šetnji duljine D u hiperkocki, s početkom u X i završetkom u Y, jednak je poziciji (X, Y) u matrici H^D.

Dokaz: Pogledajte Spielmanov knjigu/pdf o algebarskoj teoriji grafova, fantastična stvar. Za elementarni dokaz, pogledajte Lemma 3 na linku
https://courses.grainger.illinois.edu/cs598cci/sp2020/LectureNotes/lecture1.pdf □

Definicija 2. Neka je B matrica dobivena tako što od matrice H^K ostavimo samo retke i stupce koji odgovaraju dobrim bitmaskama.

Tvrdnja 3. Broj šetnji (od X do Y) duljine N u hiperkocki, gdje je N djeljiv s K, takvih da svakih K koraka budemo u dobrom stanju, jednak je polju (X, Y) u matrici $B^{N/K}$.

Dokaz: Ostavljeno za vježbu. □

Predstavljamo rješenje za teži slučaj, tj. $M = 2$. Potrebno je izračunati matricu B, potencirati je na $\lfloor N/K \rfloor$, te pomnožiti još s ostatkom koraka. Pošto je dio vezan za ostatak koraka nakon zadnjeg višekratnika od K potpuno identičan kao dio vezan za računanje matrice B, dalje opisujemo rješenje u slučaju kada je N djeljiv s K.

Potencirati matricu B je dovoljno učiniti naivnim brzim potenciranjem, množeći matrice u kubnoj složenosti, jer je za $M = 2$ ona dimenzija 100×100.

Međutim, potenciranje matrice H radi samo za $M = 1$, jer je za $M = 2$ ona oblika 1024×1024.

Za dovoljno mali K, umjesto potenciranja matrica možemo koristiti dinamiku, koja za neki početni vrh X računa broj načina za doći u svaki vrh hiperkocke u nekom broju koraka. Svako stanje dinamike utječe na 10 drugih stanja, pa je to rješenje dovoljno brzo za $K <= 15$.

Međutim, ograničenja su $K \leq N \leq 10^{18}$, što znači da je potrebno pametnije potencirati matricu hiperkocke. Ključna je sljedeća tvrdnja:

Tvrdnja 4: Element (X, Y) matrice B ovisi samo o broju bitova u kojima se X i Y razlikuju.

Dokaz: Jasno je da sve vrhove hiperkocke možemo bez smanjenja općenitost bitovno XOR-ati s nekom bitmaskom. Zato možemo pretpostaviti $X = 0$ u ovoj tvrdnji. Također, jasno je da poredak bitova ne igra nikakvu ulogu. Zato su svi Y koji imaju isti broj jedinica ekvivalentni što se tiče broja šetnji od 0 do Y. □

Tvrdnja 4 implicira da je dovoljno brzo izračunati prvi redak matrice $B = H^K$. Ovisno o implementaciji, takvo rješenje može riješiti subtask $K \leq 1500$ ili proći za sve bodove. (Natjecatelj Dorijan Lendvaj je implementirao XOR-konvoluciju, koja radi jako brzo na standardnim procesorskim arhitekturama.)

Službeno rješenje ima drukčiju ideju. Ako uvedemo simetriju i računamo za $X = 0$, dovoljno je izračunati broj šetnji koje počnu u bitcountu 0 i završe u nekom bitcountu r, za svaki $0 \leq r \leq 10$. Broj načina za

doći iz nekog bitcounta u neki drugi u jednom koraku zadan je sljedećom 11×11 matricom:

$$C = \begin{bmatrix} 0 & 10 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & 0 & 9 & 0 & \dots & 0 & 0 & 0 \\ 0 & 2 & 0 & 8 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 9 & 0 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 10 & 0 \end{bmatrix}$$

Ako potenciramo matricu C, dobit ćemo broj šetnji duljine K od bitcounta 0 do svakog drugog bitcounta. Za dovršiti rješenje, opet koristimo da je broj šetnji do svake bitmaske nekog fiksnog bitcounta jednak, pa je ukupan broj šetnji do zadanog bitcounta dovoljno podijeliti s odgovarajućim binomnim koeficijentom.

Za detalje, pogledajte izvorni kod službenog rješenja.

Postoji još brže rješenje, koje koristi *eksponencijalne funkcije izvodnice*. Ukratko, traženi broj šetnji duljine K od bitcounta 0 do neke maske s $10 - r$ bitova jednak je koeficijentu uz $\frac{x^K}{K!}$ sljedećeg izraza:

$$\left(\sum_{i=0}^{\infty} \frac{x^{2i}}{(2i)!}\right)^r \left(\sum_{i=0}^{\infty} \frac{x^{2i+1}}{(2i+1)!}\right)^{10-r} = \left(\frac{e^x + e^{-x}}{2}\right)^r \left(\frac{e^x - e^{-x}}{2}\right)^{10-r}.$$

Uz pažljivo raspisivanje, moguće je rješenje broja operacija $O((5 \cdot M)^2 + (5 \cdot M) \log K)$, to jest, dobiti elemente potencirane matrice hiperkocke nekih dimenzija $n \times n$ je moguće u $O(\log n \log K)$.

Za slične ideje, pogledajte besplatnu knjigu *generatingfunctionology* Herberta Wilfa.

Zadatak: Zagrade

Pripremila: Paula Vidas
Potrebno znanje: stog, zagrade, ad hoc

Prvo ćemo opisati rješenje u slučaju da je cijela lozinka validan niz. U prvom podzadatku je $Q = \frac{N^2}{4}$, pa možemo postaviti upit za svaki interval parne duljine (primjetimo da nema smisla pitati za intervale neparne duljine). Jedno moguće rješenje je sljedeće: pitamo za prva dva znaka, prva četiri znaka, itd. sve dok ne dobijemo pozitivan odgovor. Tada znamo na kojem je mjestu zatvorena zagrada koja je uparena s otvorenom zagradom na prvom mjestu. Sada rekurzivno riješimo isti zadatak na intervalu između tih zagrada i na intervalu desno od zatvorene zagrade.

U trećem podzadatku možemo postaviti $Q = N - 1$ upita. Koristit ćemo stog, koji je na početku prazan. Idemo redom po pozicijama lozinke. Ako je stog prazan, stavimo trenutnu poziciju na stog. Inače, postavimo upit za interval između pozicije koja je na vrhu stoga i trenutne pozicije. Ako je odgovor pozitivan, zagrade na tim pozicijama su uparene, te obrišemo vrh stoga. U suprotnom stavimo trenutnu poziciju na stog.

Što kada cijela lozinka nije validan niz? Algoritam je isti, ali na kraju stog ne mora ostati prazan. Od pozicija koje su ostale na stogu, na prvih (manjih) pola mora biti zatvorena, a na drugih pola otvorena zagrada.