Prirodno Matematički Fakultet u Nišu

Optimizacija i napredne tehnike dinamičkog programiranja

Student:
Ivan Stošić

Profesor:
Predrag Stanimirović

Niš
Februar, 2018.

Sadržaj

1 Uvod

1.1 Dinamičko programiranje

Dinamičko programiranje je tehnika rešavanja optimizacionih problema i problema prebrojavanja gde se glavni problem rešava tako što se identifikuju slični potproblemi manje veličine, koji se zatim rešavaju i čija se rešenja kombinuju u rešenje glavnog problema. Svaki ovako dobijeni potproblem se rešava najviše jedanput, nakon čega se njegovo rešenje pamti u memoriji.

Prvi korak u primeni dinamičkog programiranja na rešavanje nekog problema jeste da se identifikuju *potproblemi* tog problema. Najveća instanca među svim potproblemima jeste *glavni problem*. Najmanje instance su *trivijalni potproblemi*, koji se ne dele dalje na potprobleme i čija rešenja se dobijaju na neki drugi način. Zatim je neophodno, za svaki potproblem naći relaciju između njega i jednog ili više manjih potproblema. Ova relacija odnosno rešenje za potproblem je matematički izraz ili rezultat nekog jednostavnog algoritma u kojem figurišu rešenja manjih potproblema. Kažemo da potproblem A zavisi od potproblema B ako rešenje potproblema B figuriše u izrazu koji je rešenje potproblema A. Da bismo našli rešenje nekog potproblema neophodno je da prethodno nađemo rešenja za sve potprobleme od kojih on zavisi.

Razmotrimo sledeći primer. Neka je glavni problem nalaženje broja k-elementnih podskupova skupa $\{1, 2, \ldots, n\}$. Identifikujemo sledeće potprobleme: Za brojeve $i, j \in \mathbb{N}_0, i \leq n, j \leq k$, naći broj j-elementnih podskupova skupa $\{1, 2, \ldots, i\}$. Označimo ovaj broj sa $d(i, j)$. Glavni problem je zapravo potproblem gde je $i = n, j = k$. Trivijalni potproblemi su oni gde je $i = 0, j > 0$, u tom slučaju je $d(i, j) = 0$ i problemi gde je $j = 0$, gde važi $d(i, j) = 1$. Nađimo rešenje za sve ostale, neka su $i, j > 0$. Posmatrajmo sve j-elementne podskupove. Tada svaki od njih ili sadrži element i ili ga ne sadrži. Broj podskupova koji sadrže element i jednak je broju $(j-1)$-elementnih podskupova skupa $\{1, \ldots, i-1\}$, odnosno $d(i-1, j-1)$. Broj podskupova koji ne sadrže element i jednak je broju j-elementnih podskupova skupa od $\{1, \ldots, i-1\}$, odnosno $d(i-1, j)$. Dakle, dolazimo do sledeće formule:

$$d(i, j) = d(i-1, j) + d(i-1, j-1)$$

Ovakve formule ćemo nazivati *DP vezama*. Argument funkcije d čije vrednosti izračunavamo ćemo nazivati *DP stanjem*. U prethodno razmotrenom

primeru DP stanje je element skupa $\{0, \ldots, n\} \times \{0, \ldots, k\}$. DP stanje ne mora nužno biti uređena m-torka brojeva.

1.2 Složenost algoritma

Vreme, odnosno broj koraka i količina utrošene memorije tokom izvšenja nekog algoritma zavisi od ulaznih parametara. *Veliko O* notacija nam olakšava opisivanje i izračunavanje ovih funkecionalnih zavisnosti. Neka je u narednim definicijama domen funkcija f, g skup $\mathbb{N}_0$ a kodomen $\mathbb{R}^+ \cup \{0\}$.

Definicija 1.1 *Skup $O(g)$ definišemo kao skup svih funkcija f za koje važi da postoje konstante c i n_0 takve da je $f(n) \leq cg(n)$ za svako $n \geq n_0$.*

Ovu notaciju koristimo kada želimo da opišemo gornju granicu neke funkcije, do na proizvod sa konstantom. Često umesto $f \in O(g)$ pišemo i $f = O(g)$. Problem ove notacije je upravo u tome što samo daje gornju granicu ponašanja neke funkcije. Zato se uvodi Θ-notacija.

Definicija 1.2 *Skup $\Theta(g)$ definišemo kao skup svih funkcija f za koje važi da postoje pozitivne konstante c_1, c_2 i n_0 takve da je $c_1 g(n) \leq f(n) \leq c_2 g(n)$ za svako $n \geq n_0$.*

Za algoritam čiji je ulazni parametar n, što može biti broj elemenata niza, broj vrsta matrice, broj čvorova grafa, itd. kažemo da ima vremensku složenost $\Theta(g(n))$ ako je f, gde je $f(n)$ broj elementarnih koraka tokom izvršenja algoritma, u skupu $\Theta(g)$. Slično definišemo memorijsku složenost preko broja iskorišćenih elementarnih memorijskih lokacija npr. bajtova ili bitova.

2 Optimizacija DP algoritama

Algoritmi dinamičkog programiranja mogu da se optimizuju u zavisnosti od problema i DP veze. U ovom odeljku razmotrićemo neke od strategija optimizacije.

2.1 Optimizacija memorije

Da bi se izračunalo rešenje glavnog problema neophodno je izračunati rešenja nekih potproblema. Međutim, vrlo često je slučaj da nam nisu u svakom trenutku potrebna sva rešenja do tog trenutka izračunatih potproblema.

Razmotrimo ponovo primer dat u uvodnoj sekciji. Skup stanja je $\{0, \ldots, n\} \times \{0, \ldots, k\}$ a DP veza je

$$d(i, j) = d(i-1, j) + d(i-1, j-1)$$

Primetimo da je broj iskorišćenih memorijskih lokacija $\Theta(nk)$. Ako posmatramo DP vezu, uočavamo da sve vrednosti $d(n, \cdot)$ možemo izračunati ako poznajemo sve vrednosti $d(n-1, \cdot)$. Ova činjenica nas navodi na sledeću strategiju optimizacije: U svakom trenutku držimo samo vrednosti $d(i, \cdot)$ i $d(i-1, \cdot)$, odnosno samo dve susedne vrste matrice. Sledi efikasna implementacija ovog poboljšanog algoritma. Nadalje, sve implementacije biće date u programskom jeziku C++14, ako nije drugačije navedeno. Pretpostavlja se da su uključene odgovarajuće standardne biblioteke.

```
int brojSkupova(int n, int k) {
    vector<int> d[2];
    d[0].resize(k+1, 0);
    d[1].resize(k+1);
    d[0][0] = 1;
    for (int i=1; i<=n; i++) {
        d[i%2][0] = 1;
        for (int j=1; j<=k; j++)
            d[i%2][j] = d[(i-1)%2][j] + d[(i-1)%2][j-1];
    }
    return d[n%2][k];
}
```

U opštem slučaju, želimo da podelimo sva DP stanja na m slojeva, gde će za izračunavanje rešenja u i-tom sloju biti dovoljne vrednosti samo iz slojeva $i, i-1, \ldots, i-l$, gde je l mala konstanta. U tom slučaju memorijska složenost se može smanjiti na $O(P)$, gde je P maksimalni broj elemenata u jednom sloju. U prethodnom primeru koristimo samo dva sloja, svaki sloj se sastoji od $k+1$ elemenata pa je memorijska složenost $O(k)$, odnosno tačno $\Theta(k)$.

2.2 Upotreba pomoćnih struktura podataka

Kod pojedinih problema izraz u DP vezi može uključivati veliki broj manjih DP stanja, odnosno, njihov broj može biti i nekonstantan. Primer je problem nalaženja najvećeg rastućeg podniza. Neka je ulaz $a_1, \ldots, a_n$ niz različitih brojeva. Potrebno je naći podskup $I = \{i_1, i_2, \ldots, i_k\}$ skupa $\{1, \ldots, n\}$ takav da se maksimizuje $|I| = k$ i važi $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$ i $i_1 < i_2 < \ldots < i_k$.

Ovaj problem ima jednostavno DP rešenje. Neka je d_i veličina najvećeg skupa I koji sadrži element i i neke od elemenata iz skupa $\{1, \ldots, i-1\}$. Rešenje glavnog problema je onda $\max\{d_1, \ldots, d_n\}$, pošto se svaki podniz mora završavati na nekoj od ovih n pozicija. Postoji jedan trivijalni potproblem $d_1 = 1$, a za $i > 1$ važi:

$$d_i = 1 + \max_{j<i, a_j<a_i} \{d_j\} \tag{2.2.1}$$

Ukoliko takvo j ne postoji, onda je $d_i = 1$. Primetimo da direktna implementacija ove DP veze daje algoritam vremenske složenosti $\Theta(n^2)$.

Međutim, možemo iskoristiti augmentovano balansirano binarno stablo da ubrzamo izračunavanje izraza u jednačini (2.2.1). Pomenuta struktura podataka se ponaša kao sortirani rečnik odnosno sortirani niz parova (k_i, v_i), gde je k jedinstveni ključ a v uskladištena vrednost. Stablo je augmentnovano dodatnom operacijom `getRange(l, r)` koja nalazi vrednost agregatne funkcije $f(v_x, \ldots, v_y)$ na proizvoljnom zadatom opsegu ključeva $l \leq k_x, \ldots, k_y < r$. U našem slučaju agregatna funkcija je maksimum brojeva. Sve operacije (ubacivanje, brisanje, `getRange`) rade u složenosti $O(\log W)$, gde je W trenutni broj elemenata u stablu. U trenutku pre računanja vrednosti d_i u stablu čuvamo parove (a_j, d_j) za sve $j < i$. Zatim, pozovemo funkciju `getRange` sa argumentima $l = -\infty, r = a_i$. Ova funkcija nam vraća maksimalnu uskladištenu vrednost (drugi element uređenog para) za sve ključeve koji su manji od a_i, odnosno vraća upravo $q = \max_{j<i, a_j<a_i}\{d_j\}$. Dodatno proveravamo da li je ovaj skup ključeva prazan, u tom slučaju postavljamo $q = 0$. Konačno, postavljamo $d_i = q + 1$ u skladu sa (2.2.1). Nakon izračunavanja d_i dodajemo par (a_i, d_i) u stablo. Kako operacije dodavanja u stablo i `getRange` imaju vremensku složenost $O(\log n)$, ceo algoritam ima složenost $O(n \log n)$.

2.3 Optimizacija pomoću linearnih funkcija

Posmatrajmo jedodimenzione probleme kod kojih DP veza uključuje sve prethodne potprobleme. Neka je DP veza takva da se može svesti na traženje vrednosti $\max_{j<i}\{f_j(x_i)\}$ (ili minimuma), gde su f_j linearne funkcije, čiji koeficijent pravca i konstanta se mogu dobiti tek nakon izračunavanja vrednosti d_j.

Sledi primer ovakvog problema. Neka postoji n trkača, koji u početku svi stoje u mestu na pozicijama $0 < x_1 < x_2 < \ldots < x_n$. Za svakog trkača

poznato je vreme potrebno da pređe jedinično rastojanje s_i i vreme zagrevanja t_i - ovo je vreme koje mu je potrebno od trenutka kada primi štafetu do trenutka kada počne da trči. Nijedan trkač ne sme da trči ukoliko ne nosi štafetu i svaki trkač može trčati samo ulevo. U početku se štafeta nalazi kod n-tog trkača. Koliko je najmanje vremena potrebno da se štafeta dostavi na poziciju $x = 0$?

Neka je d_i minimalno vreme potrebno da i-ti trkač dostavi štafetu na poziciju $x = 0$ od trenutka kada primi štafetu. Dodatno definišemo trivijalni potproblem $d_0 = 0$ i postavljamo $x_0 = 0, t_0 = 0$. Neka je $i > 0$. Tada i-ti trkač treba da izabere kom trkaču će dostaviti štafetu, ili će je odneti skroz do cilja. Oba slučaja opisana su jednom DP vezom:

$$d_i = t_i + \min_{0 \le j < i} \{d_j + s_i(x_i - x_j)\} \tag{2.3.1}$$

Glavni problem je računanje d_n. Ponovo, pravolinijska implementacija algoritma koji koristi ovu DP vezu ima vremensku složenost $\Theta(n^2)$. Zapišimo jednačinu (2.3.1) na drugačiji način:

$$d_i = t_i + \min_{0 \le j < i} \{d_j + s_i x_i - s_i x_j\}$$

Kako $s_i x_i$ ne zavisi od j može da izađe ispred minimuma:

$$d_i = t_i + s_i x_i + \min_{0 \le j < i} \{d_j - s_i x_j\}$$

Odnosno, ako definišemo $f_j(z) = d_j - z x_j$:

$$d_i = t_i + s_i x_i + \min_{0 \le j < i} \{f_j(s_i)\}$$

Sada smo problem sveli na održavanje skupa Q linearnih funkcija, sa operacijom dodavanja linearne funkcije i traženjem $\min_{f \in Q} f(x)$ za zadato x. Održavaćemo za svaku funkciju f_i u skupu Q, skup brojeva $M_i \subseteq \mathbb{R}$ za koji važi da je $f_i(x)$ najmanja vrednost od svih $f_j(x), f_j \in Q$.

Dokažimo da je skup M_i jednak nekom, možda beskonačnom ili praznom intervalu u $\mathbb{R}$. Označimo sa $P_{i,j}$ skup svih realnih brojeva x takvih da je $f_i(x) < f_j(x)$. Kako su funkcije f_i, f_j linearne, napišimo ih u obliku $f_i(x) = k_i x + b_i, f_j(x) = k_j x + b_j$. Ukoliko je $k_i = k_j$, imamo dva podslučaja: Ako je $b_i < b_j$, tada je za svako $x, f_i(x) < f_j(x)$ pa je $P_{i,j} = (-\infty, +\infty)$. U suprotnom je $P_{i,j} = \emptyset$. Neka je sada $k_i \neq k_j$. Tada je $f_i(x) = f_j(x)$ za $x = \frac{b_j - b_i}{k_i - k_j} = x_0$. Ukoliko je $k_i < k_j$, $P_{i,j}$ biće jednak $(-\infty, x_0)$, u suprotnom

$(x_0, +\infty)$, čime smo dokazali tvrđenje.

Kako je $M_i = \bigcap_{j \neq i} P_{i,j}$, odnosno presek konačno mnogo intervala, i M_i je interval. Primetimo i da su intervali M_i disjunktni. Iz ovog razloga možemo da čuvamo sve neprazne intervale M_i u rastućem redosledu po levom kraju ili po desnom (svejedno je, pošto se ne seku). Primetimo da se u ovakvom uređenju M_i može naći pre M_j samo ukoliko je $k_i > k_j$. Takođe, primetimo da je $\mathbb{R} \setminus \bigcup M_i$ konačan skup, odnosno, unija M_i pokriva celu realnu pravu osim u konačno mnogo tačaka.

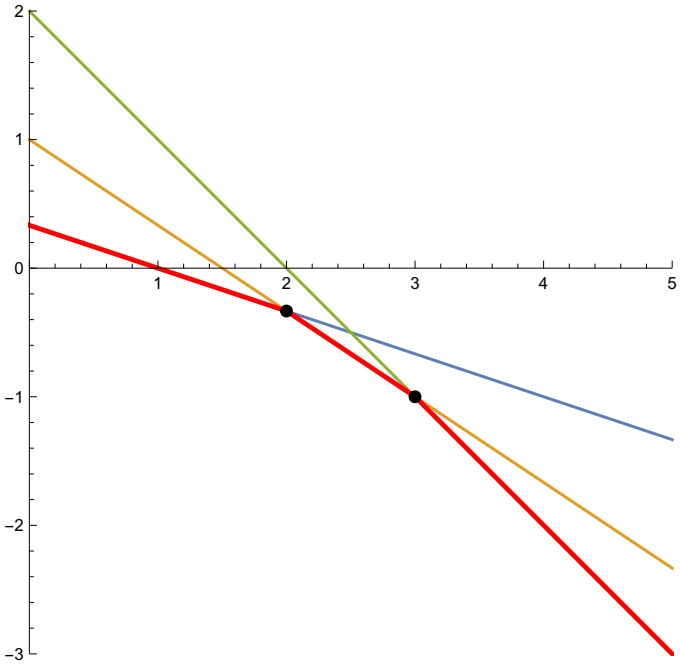

Figura 2.3.1: Prikaz intervala minimalnosti M_i

Opišimo sada algoritam za nalaženje $\min_{f \in Q}\{f(x)\}$. Jednostavno, binarnom pretragom nalazimo interval M_i u nizu sortiranih intervala i vraćamo $f_i(x)$. Složenost je $O(\log |Q|)$.

Opišimo zatim algoritam za dodavanje nove funkcije $f_i(x) = k_i x + b_i$. U opštem slučaju, algoritam je složeniji i zahteva da se kolekcija Q čuva u balansiranom binarnom stablu. Opisaćemo jednostavniju verziju gde pretpostavljamo da je $k_i < k_j$ za svako $f_j(x) = k_j x + b_j$ gde je $f_j \in Q$. Drugim rečima, dodajemo funkcije u opadajućem redosledu koeficijenata pravaca. Kod velikog broja problema koji se rešavaju ovom metodom važi upravo ova pretpostavka – takav je i problem koji je dat na početku sekcije – koeficijent

pravca i-te funkcije koja se dodaje je $-x_i$, a x_i je rastući niz.

Algoritam je jednostavan: računa se presek nove funkcije sa starim intervalima M_i i ovi intervali se po potrebi smanjuju ili izbacuju. U slučaju da nova funkcija ima manji koeficijent pravca od svih trenutno ubačenih funkcija, svi promenjeni ili izbačeni intervali će se nalaziti na kraju uređenog niza, i pritom će najviše jedan biti smanjen, dok će svi ostali (desno od njega) biti izbačeni. Dakle, algoritam se sastoji od toga da izbacujemo intervale sa desnog kraja dokle god se oni celi nalaze iznad nove funkcije, a prvi put kada interval seče novu funkciju, zamenimo taj interval presekom i dodamo novu funkciju sa desne strane, ka $+\infty$. Vremenska složenost je $O(n)$ – i-ta operacija dodavanja sastoji se od $O(z_i)$ koraka, gde je z_i broj izbačenih intervala u i-tom koraku. Kako svaki interval može biti izbačen najviše jednom, vremenska složenost je $O(n)$.

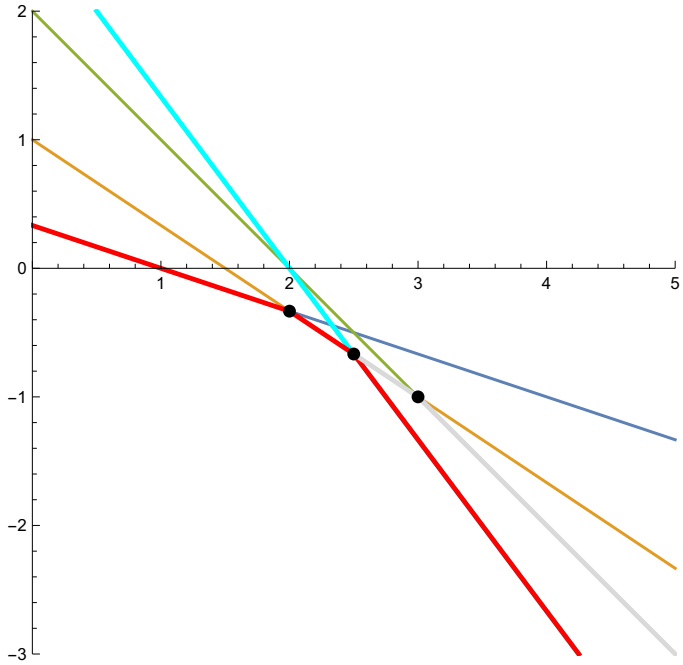

Figura 2.3.2: Dodavanje nove funkcije. Nova funkcija obojena je cijan. Izbačeni intervali su svetlo-sivi

Izračunajmo i vremensku složenost. Imamo $O(n)$ binarnih pretraga, svaka ima vremensku složenost $O(\log n)$ i pomenuto je da sva dodavanja zajedno imaju ukupnu vremensku složenost $O(n)$, pa je konačna vremenska složenost $O(n \log n)$, što je značajno ubrzanje u odnosu na polaznu složenost $\Theta(n^2)$.

2.4 Podeli-pa-vladaj optimizacija

Posmatrajmo probleme kod kojih je data funkija cene $c(i,j), i \leq j$, za koju važi $c(i,i) = 0$ i $c(i,j) > 0$, kada je $i < j$. (koristićemo i zapis $c_{i,j}$) i broj k, a potrebno je naći niz indeksa $0 \leq x_1 \leq x_2 \leq \ldots \leq x_k \leq n$ takav da se minimizuje $c(0,x_1) + c(x_1,x_2), \ldots + c(x_{k-1},x_k) + c(x_k,n)$. Jasno je da se ovaj problem u opštem slučaju može rešiti sledećim algoritmom dinamičkog programiranja:

Neka je $d_{i,j}$ minimalni zbir koji se može dobiti od svih izraza oblika $c(0,x_1)+c(x_1,x_2)+\ldots+c(x_{i-1},x_i)+c(x_i,j)$. Rešićemo $d_{i,j}$ za svako $0 \leq i \leq k$ i $0 \leq j \leq n$. Glavni problem je nalaženje vrednosti $d(k,n)$. Trivijalni potproblemi su oni gde je $i = 0$ i tada je $d_{0,j} = c_{0,j}$. Za $i > 0$ važi sledeća DP veza:

$$d_{i,j} = \min_{0 \leq l \leq j} \{d_{i-1,l} + c_{l,j}\} \tag{2.4.1}$$

Postoji $O(nk)$ potproblema i svaki rešavamo u vremenskoj složenosti $O(n)$, pa je konačna vremenska složenost algoritma koji direktno implementira ovu DP vezu $O(n^2k)$.

Međutim, ukoliko funkcija cene $c(i,j)$ zadovoljava određene uslove, moguće je rešavati potprobleme u boljoj vremenskoj složenosti. Jedna takva osobina je monotonost: Funkcija c je monotona ako važi $c(i,j) \leq c(i,k)$ i $c(j,k) \leq c(i,k)$ za svako $i < j < k$. Drugim rečima, ukoliko se "opseg" funkcije proširi, vrednost ne može da se smanji. Drugi uslov koji je od velikog značaja je tzv. nejednakost četvorougla. Kažemo da funkcija c zadovoljava ovu nejednakost ako važi:

$$c(i,k) + c(j,l) \leq c(i,l) + c(j,k) \tag{2.4.2}$$

za svako $i \leq j \leq k \leq l$. Primer funkcije koja je monotona i zadovoljava nejednakost četvorougla je $c(i,j) = (a_i + a_{i+1} + \ldots + a_{j-1})^2$, gde su a_i dati pozitivni brojevi. Iz nejednakosti četvorougla sledi monotonost: ako je $i < j < k$, imamo:

$$c(i,j) + c(j,k) \leq c(j,j) + c(i,k)$$

pošto je $c(j,j) = 0$ i $c(j,k) \geq 0$, direktno dobijamo

$$c(i,j) \leq c(i,k)$$

a iz $c(j,j) = 0$ i $c(i,j) \geq 0$ dobijamo

$$c(j,k) \leq c(i,k).$$

Dakle, u određenom smislu, nejednakost četvorougla uopštava monotonost.

Primetimo da se izračunavanje DP veze (2.4.1) može izvesti u k koraka, gde se u i-tom koraku računaju vrednosti $d_{i,\cdot}$, i svaki korak ima vremensku složenost $O(n^2)$. Ukoliko funkcija c zadovoljava nejednakost četvorougla, jedan sloj se može izračunati u vremenskoj složenosti $O(n \log n)$, na sledeći način. Definišimo $p_{i,j}$ kao vrednost indeksa l za koju se dostiže minimalna vrednost u izrazu (2.4.1). Ukoliko ima više takvih vrednosti, uzimamo najmanju. Ključna observacija je da za $i > 0$ i $x < y$ važi $p_{i,x} \leq p_{i,y}$.

Dokažimo ovo tvrđenje. Pretpostavimo suprotno: $p_{i,x} > p_{i,y}$. Uvedimo oznake: $\alpha = p_{i,x}$ i $\beta = p_{i,y}$. Primenimo nejednakost četvorougla na indekse $\beta < \alpha \leq x < y$:

$$c(\beta, x) + c(\alpha, y) \leq c(\alpha, x) + c(\beta, y)$$

Po definiciji α kao najmanjeg indeksa koji minimizuje $d_{i-1,\alpha} + c(\alpha, x)$ važi:

$$d_{i-1,\alpha} + c(\alpha, x) < d_{i-1,\beta} + c(\beta, x)$$

Ako saberemo ovu nejednakost sa prethodnom, nakon kraćenja dobijamo

$$d_{i-1,\alpha} + c(\alpha, y) < d_{i-1,\beta} + c(\beta, y)$$

što je u kontradikciji sa izborom β.

Algoritam koji je opisan u nastavku se oslanja na ovu osobinu monotonosti indeksa $p_{i,j}$. Algoritam odjednom nalazi sve vrednosti $p_{i,j}$, samim tim i $d_{i,j}$, za neki opseg vrednosti $j \in [l, r] = \{l, l+1, \ldots, r\}$. Implementacija je rekurzivna, kreće se od opsega $[l, r] = [0, n]$. Svaki rekurzivni poziv sadrži i informaciju o tome u kom opsegu $[u, v]$ se mogu kretati vrednosti $p_{i,j}, j \in [l, r]$. Zatim nalazimo vrednost $w = p_{i,m}$, gde je $m = \left\lfloor \frac{l+r}{2} \right\rfloor$, pretragom po celom opsegu $[u, v]$, i odmah izračunavamo $d_{i,m} = d_{i-1,w} + c(w, m)$. Iz monotonosti $p_{i,j}$ znamo da su sve vrednosti $p_{i,j}$ gde je $j \in [l, m-1]$ iz opsega $[u, w]$ i slično da su sve vrednosti $p_{i,j}$ gde je $j \in [m+1, r]$ iz opsega $[w, v]$. Sledi generička implementacija algoritma:

```
void resiSloj(auto& d, auto c, int i,
          int l, int r, int u, int v) {
    if (l > r)
        return;
    int m = (l+r) / 2, w = u;
    for (int j=u+1; j<=v && j<=m; j++)
        if (d[i-1][j] + c(j, m) < d[i-1][w] + c(w, m))
            w = j;
    d[i][m] = d[i-1][w] + c(w, m);
    resiSloj(d, c, i, l, m-1, u, w);
    resiSloj(d, c, i, m+1, r, w, v);
}

void resiSve(auto& d, auto c, int n, int k) {
    for (int j=0; j<=n; j++)
        d[0][j] = c(0, j);
    for (int i=1; i<=k; i++)
        resiSloj(d, c, i, 0, n, 0, n);
}
```

Ovde je d objekat poput dvodimenzionalne matrice u koji mogu da se upisuju vrednosti, c je *callback* funkcija koja izračunava cenu. Izračunajmo složenost ovog algoritma. Pretpostavljamo da se funkcija c izvršava u vremenskoj složenosti $O(1)$. Složenost je jednaka k puta složenost izvršavanja jednog sloja – nađimo ovu složenost. Kako se u svakom rekurzivnom pozivu funkcije `resiSloj` segment $[l, r]$ polovi, dubina stabla rekurzije je $O(\log n)$. Posmatrajmo zajedno sve pozive na i-tom nivou rekurzije. Ovaj nivo se sastoji od ne više od 2^i poziva – označimo ovaj broj sa g. Za vrednosti u_j, v_j unutar jednog sloja važe sledeće nejednakosti:

$$u_1 \leq v_1 \leq u_2 \leq v_2 \leq \ldots u_g \leq v_g$$

Kako je vremenska složenost izvršenja koda unutar jednog poziva funkcije (bez rekurzivnih poziva) jednaka $O(v-u)$, u zbiru, vremenska složenost biće jednaka $O(\sum(v_i-u_i)+g) = O(n+g) = O(n)$. Kako ima $\log n$ nivoa, ukupna složenost jednog sloja je $O(n \log n)$, a celog algoritma $O(kn \log n)$.

2.5 Knutova optimizacija

Posmatrajmo probleme čija je DP veza oblika:

$$d_{i,j} = \min_{i<k<j} \{d_{i,k} + d_{k,j} + c(i,j)\} \tag{2.5.1}$$

za $0 \leq i, j \leq n$ i $j - i \geq 2$. Trivijalni potproblemi su $d_{i,i+1} = c(i, i+1)$ za $0 \leq i < n$ a cilj nam je da izračunamo vrednost $d_{0,n}$. Direktna implementacija algoritma koji koristi ovu DP vezu ima vremensku složenost $\Theta(n^3)$, dok je memorijska složenost $\Theta(n^2)$. Ukoliko za funkciju c važi nejednakost četvorougla (2.4.2), moguće je ubrzati algoritam. Označimo sa $p_{i,j}$ najmanju vrednost k koja minimizuje izraz (2.5.1). Tada važi sledeća nejednakost [1]:

$$p_{i,j-1} \leq p_{i,j} \leq p_{i+1,j} \tag{2.5.2}$$

Ovo nam omogućava da za svako i, j vrednost k tražimo na manjem opsegu. Sledi generička implementacija poboljšane verzije algoritma.

```
void resiKnuth(auto& d, auto& p, auto c, int n) {
    for (int i=0; i<n; i++) {
        d[i][i+1] = c(i, i+1);
    }
    for (int i=n-2; i>=0; i--) {
        d[i][i+2] = c(i, i+1) + c(i+1, i+2) + c(i, i+2);
        p[i][i+2] = i+1;
        for (int j=i+3; j<=n; j++) {
            int& w = p[i][j] = p[i][j-1];
            for (int k = p[i][j-1]+1; k <= p[i+1][j]; k++)
                if (d[i][k] + d[k][j] < d[i][w] + d[w][j])
                    w = k;
            d[i][j] = d[i][w] + d[w][j] + c(i, j);
        }
    }
}
```

Slično kao u prethodnoj sekciji, d, p su objekti poput dvodimenzionalne matrice u koje mogu da se upisuju vrednosti, dok je c je *callback* funkcija koja izračunava cenu. Memorijska složenost ostaje $\Theta(n^2)$. Izračunajmo vremensku složenost ovog algoritma. Grupišimo iteracije spoljne dve for-petlje (i, j) po vrednosti $g = j - i$. Vreme izvršenja za $g = 1$ je $O(n)$. Za $g \geq 2$ u vremenu izvršenja dominira unutrašnja for-petlja. Ukupan broj iteracija

koje izvrši ova for-petlja u okviru grupe g iznosi

$$\begin{aligned} & \sum_{i=0}^{n-g}(p_{i+1,i+g} - p_{i,i+g-1}) \\ = & \sum_{i=0}^{n-g} p_{i+1,i+g} - \sum_{i=0}^{n-g} p_{i,i+g-1} \\ = & \sum_{i=1}^{n-g+1} p_{i,i+g-1} - \sum_{i=0}^{n-g} p_{i,i+g-1} \\ = & \; p_{n-g+1,n} - p_{0,g-1} \\ \leq & \; n - 0 \end{aligned} \tag{2.5.3}$$

što je $O(n)$. Dakle, ukupna vremenska složenost izvršenja je $O(n^2)$, odnosno, pošto svakako imamo dve for-petlje po i, j, tačno $\Theta(n^2)$.

Primer problema čija DP veza ima oblik *nalik na* (2.5.1) jeste problem nalaženja optimalnog binarnog stabla pretrage: Postoji n jedinstvenih ključeva, uređenih u rastućem redosledu – i-tom ključu ($1 \leq i \leq n$) pridružena je verovatnoća p_i da će u stablu biti tražen baš taj ključ. Važi $\sum_{i=1}^{n} p_i = 1$. Potrebno je pronaći binarno stablo pretrage koje minimizuje očekivano vreme traženja ključa, odnosno, ako je čvor i na dubini $b_i \geq 1$, treba minimizovati izraz $\sum_{i=1}^{n} b_i p_i$. Definišimo $d_{i,j}$ kao minimalno očekivano vreme traženja ključeva ako formiramo samo stablo od ključeva čiji su redni brojevi $i, i+1, \ldots, j$. Za $i = j$ važi $d_{i,j} = p_i$ – koren se nalazi na dubini 1. Za $d_{i,j}$ gde je $i > j$ definišemo vrednost 0 – ovaj slučaj odgovara stablu bez čvorova. U opštem slučaju, od čvorova $i, i+1, \ldots, j$ treba odabrati koren k i rasporediti ostale čvorove u levo i desno podstablo. Čvorove sa indeksima manjim od k moramo staviti u levo, a ostale u desno podstablo da bismo dobili binarno stablo pretrage. Dakle, treba minimizovati vrednost izraza

$$\sum_{u=i}^{k-1} b_u p_u + p_k + \sum_{u=k+1}^{j} b_u p_u \tag{2.5.4}$$

Kako je dubina čvorova u levom podstablu za 1 veća od dubine koju bi ti čvorovi imali u slučaju da je to celo stablo, suma sa leve strane $\sum_{u=i}^{k-1} b_u p_u$ je za tačno $\sum_{u=i}^{k-1} p_u$ veća od sume koja bi se dobila u slučaju da je to celo stablo, pa isti raspored čvorova istovremeno minimizuje oba izraza. Iz ovoga sledi da je zapravo minimalna vrednost sume $\sum_{u=i}^{k-1} b_u p_u$ jednaka $d_{i,k-1} + \sum_{u=i}^{k-1} p_u$,

pa je minimalna vrednost izraza (2.5.4) jednaka

$$d_{i,j} = \min_{i \le k \le j} \{d_{i,k-1} + d_{k+1,j} + \sum_{u=i}^{k-1} p_u + \sum_{u=k+1}^{j} p_u + p_k\}$$

odnosno

$$d_{i,j} = \min_{i \le k \le j} \{d_{i,k-1} + d_{k+1,j} + \sum_{u=i}^{j} p_u\} \tag{2.5.5}$$

Primetimo da ova DP veza nema oblik identičan obliku (2.5.1). Srećom, za ovaj problem važi nejednakost (2.5.2) [2], pa je gore opisani algoritam i dalje primenljiv, uz manje modifikacije. Funkciju $c(i,j) = \sum_{u=i}^{j} p_u$ možemo računati u vremenskoj složenosti $O(1)$ pomoću pomoćnog niza *prefiksnih suma*.

2.6 Binarno stepenovanje

Posmatrajmo probleme kod kojih DP veza ima sledeći oblik:

$$d_{i,j} = \sum_{k=1}^{n} A_{j,k} \cdot d_{i-1,k} \tag{2.6.1}$$

za $1 \le j \le n$, $1 \le i \le m$, a trivijalni potproblemi su oni gde je $i = 0, 1 \le j \le n$, gde je $d_{0,j} = b_j$. Cilj nam je da izračunamo vrednosti $d_{m,\cdot}$. Postoji nm netrivijalnih vrednosti i svaku računamo u složenosti $O(n)$, pa je ukupna vremenska složenost direktne implementacije $O(n^2 m)$. Memorijska složenost (ne računajući matricu a) se može smanjiti na $O(n)$ pomoću tehnike optimizacije memorije opisane u sekciji 2.1. Jasno je da DP veza (2.6.1) odgovara proizvodu matrice A i vektora d_{i-1}:

$$d_i = A \cdot d_{i-1}$$

Kako je $d_0 = b$ i množenje matrica je asocijativno, važi

$$d_i = A^i \cdot b$$

Moguće je primeniti tehniku binarnog stepenovanja da bi se direktno izračunala matrica A_m, a samim tim i vektor $d_m = A_m \cdot b$. Naime, m-ti stepen ($m \in \mathbb{N}$) nekog broja, matrice, ili elementa proizvoljne polugrupe a^m se može izračunati pomoću sledećeg rekurzivnog algoritma:

- Ukoliko je $x = 1$, vrati a.
- Ukoliko je x paran broj, izračunaj $q = a^{\frac{x}{2}}$ i vrati q^2
- Ukoliko je $x > 1$ neparan broj, izračunaj $q = a^{x-1}$ i vrati aq.

Kako posle svaka dva rekurzivna koraka vrednost postane bar duplo manja, broj množenja je $O(2 \log_2 m) = O(\log m)$. Množenje dve matrice se može realizovati u vremenskoj složenosti $O(n^3)$ naivnim algoritmom, a postoji i algoritam složenosti $O(n^{2.3729})$ [4], tako da je konačna vremenska složenost $O(n^{2.3729} \log m)$, što je pogodnije za veoma velike vrednosti m.

Primer problema koji se može rešiti na ovaj način je problem brojanja šetnji dužine m u grafu zadatog matricom susedstva A, od čvora u do čvora v. Za početni vektor b uzimamo $b_u = 1$ i $b_i = 0, i \neq u$. Konačno rešenje je polje $d_{m,v}$ vektora $d_m = A^m \cdot b$.

3 Napredne DP tehnike

U ovoj glavi bavićemo se problemima čija formulacija preko dinamičkog programiranja nije očigledna, kao i problemima kod kojih manje očigledna formulacija daje daleko efikasniji algoritam.

3.1 DP sa nekompletnim stanjima

Posmatrajmo sledeći problem:

Postoji n studenata, i-ti ($1 \leq i \leq n$) ima efikasnost a_i, i dat je broj k. Potrebno je naći broj particija skupa studenata tj. broj familija P tako da važi:

- $\bigcup_{X \in P} X = \{1, \ldots, n\}$
- $\forall X, Y \in P, X \neq Y \implies X \cap Y = \emptyset$
- $\sum_{X \in P} \delta(X) \leq k$

gde je $\delta(X)$ disbalans skupa studenata, $\delta(X) = \max_{i \in X}\{a_i\} - \min_{i \in X}\{a_i\}$. Vrednosti a_i su prirodni brojevi i važi $a_i \leq A$.

Za početak, sortirajmo niz a. Neka je $d(i, j, l)$ broj takvih particija koristeći prvih i studenata, takvih da ima ukupno j *nedovršenih* skupova i da je

ukupna *projektovana* suma disbalansa jednaka l. Ukoliko nema nedovršenih skupova, ova projektovana suma disbalansa jednaka je ukupnoj sumi disbalansa (za sve dovršene tj. kompletne skupove). Svaki nedovršen skup je neprazan i doprinosi projektovanoj sumi disbalansa sa $-a_x$, gde je x indeks prvog studenta koji je dodat u taj skup. Pošto su studenti sortirani, ovo će upravo biti minimalno a_x tog skupa. U i-tom potezu možemo izabrati jednu od četiri opcije:

1. Započinjemo nov nekompletan skup. Ukupna projektovana suma (vrednost l) se povećava za $-a_i$, a j (broj nekompletnih skupova) se povećava za jedan.

2. Zatvaramo neki ranije otvoren skup. Kako je a_i najveća vrednost tog skupa, l se povećava za a_i a broj otvorenih skupova se smanjuje za jedan. Ovo možemo učiniti samo ukoliko je $j > 0$.

3. Dodajemo a_i u novi, zaseban skup koji odmah proglašavamo zatvorenim. Vrednosti j, l se ne menjaju.

4. Dodajemo a_i u neki od trenutno otvorenih skupova, bez zatvaranja. Ovo možemo uraditi na j različitih načina. Vrednosti j, l se ne menjaju.

Obrazložimo stavku 2. Recimo da i-tog studenta dodajemo u neki ranije otvoren skup čija je minimalna vrednost a_x. U trenutku kada smo otvorili taj skup, smanjili smo l za a_x. U trenutku zatvaranja povećali smo l za a_i, pa smo efektivno povećali l za $a_i - a_x$, što je upravo disbalans ovog skupa.

Nas na kraju zanima vrednost $\sum_{l=0}^{k} d(n, 0, l)$ (na kraju ne sme biti nedovršenih skupova). Kako j može uzimati vrednosti $0 \leq j \leq n$ a l može uzimati vrednosti $-nA \leq l \leq nA$, ima ukupno $O(n^3 A)$ stanja. Svako rešavamo u $O(1)$ pa je ukupna vremenska složenost $O(n^3 A)$ što čini ovaj algoritam pseudopolinomijalnim, slično kao poznati algoritmi za problem ranca.

3.2 DP na povezanim komponentama

Ova tehnika i njene modifikacije se mogu koristiti kod zadataka koji zahtevaju brojanje permutacija koje zadovoljavaju neku osobinu. Primer je sledeći zadatak:

Dat je skup različitih celih brojeva $\{a_1, \ldots, a_n\}$ i prirodan broj k. Koliko ima permutacija $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ tako da važi:

$$\sum_{i=1}^{n} |a_{\sigma_i} - a_{\sigma_{i+1}}| = k$$

U ovoj sumi uzimamo da je $\sigma_{n+1} = \sigma_1$. Za brojeve a_i važi $0 \leq a_i \leq A$.

Slično kao u primeru iz prethodne sekcije, želimo da nađemo rešenje pseudopolinomijalne vremenske složenosti, odnosno polinomijalne po n, A. Definisaćemo DP stanje na sledeći način: neka je $d_{i,j,l}$ broj skupova koji opisuju povezane komponente delimično popunjenih permutacija gde je ubačeno i brojeva, ima j povezanih komponenti i ukupna projektovana suma je l. Redosled povezanih komponenti u permutaciji nije bitan ali redosled elemenata u povezanoj komponenti jeste. Delimično popunjena permutacija je svaki niz od n elemenata od kojih su i poznati, a ostali su nepoznati. Povezana komponenta je niz uzastopnih poznatih vrednosti – na primer, $(1, ?, 2, ?, 3)$ ima dve povezane komponente jer uzimamo da su $3, 1$ susedni. Ovo je posledica oblika sume. Dalje, za projektovanu sumu uzimamo sumu apsolutnih razlika za sve susedne parove poznatih brojeva i dodajemo -1 puta suma svih poznatih brojeva koji se nalaze na rubu povezane komponente. Ukoliko se poznat broj nalazi u komponenti koja sadrži samo njega (nema suseda), onda taj broj računamo dvaput. Ključna ideja jeste da brojeve ubacujemo u permutaciju u rastućem poretku. Posmatrajmo moguće situacije koje mogu da nastanu dodavanjem novog, $(i+1)$-og broja.

1. Broj je dodat u novu povezanu komponentu i nema suseda. Ovo je moguće uraditi na jedan način ukoliko je $n - i > 2j$, inače nije moguće. Ima $n-i$ nepopunjenih mesta od kojih su tačno $2j$ susedi neke postojeće komponente. Vrednost j se povećava za 1 dok se suma l povećava za $-2a_{i+1}$.

2. Broj je dodat na rub jedne komponente, ali njegovim dodavanjem nisu spojene dve različite komponente. Ovo je moguće uraditi na $2j$ načina ukoliko je $n - i > j$. Broj povezanih komponenti se ne menja. Neka je a_r vrednost broja koji je bio na rubu gde je dodat broj a_{i+1}. Od sume prvo oduzimamo $-a_r$ odnosno dodajemo a_r. Zatim dodajemo $|a_{i+1} - a_r| = a_{i+1} - a_r$, jer brojeve ubacujemo u rastućem redosledu i konačno dodajemo $-a_{i+1}$ jer je sada ovaj broj na rubu. Ukupno, suma će se promeniti za $a_r + a_{i+1} - a_r - a_{i+1} = 0$.

3. Broj spaja dve povezane komponente. Ovo je moguće uraditi na $j(j-1)$ načina – to je proizvod broja načina da izaberemo prvu (sa leve strane) i drugu (sa desne strane), redosled je važan. Mora važiti $j > 1$. Broj povezanih komponenti se smanjuje za 1. Ako su a_p i a_q rubni elementi te dve komponente, prvo od sume oduzimamo $-a_p$ i $-a_q$ odnosno dodajemo $a_p + a_q$. Zatim dodajemo $|a_{i+1} - a_p| + |a_{i+1} - a_q|$ odnosno

$2a_{i+1} - a_p - a_q$. Ne javljaju se novi rubni elementi. Ukupno, suma l se povećava za $2a_{i+1}$.

Ciljno stanje je ono gde je $i = n, j = 1, l = k$. Računamo $d_{i,j,l}$ u rastućem redosledu po i, rukovodeći se gore opisanim postupkom. Trivijalan potproblem, odnosno polazna vrednost je $i = 0, j = 0, l = 0$, gde je $d_{i,j,l} = 1$, postoji samo jedan prazan raspored.

Kako i uzima vrednosti iz skupa $\{0, \ldots, n\}$, j iz skupa $\{0, \ldots, \lfloor \frac{n}{2} \rfloor\}$, a l iz skupa $\{-2nA, \ldots, 2nA\}$, ima ukupno $O(n^3A)$ stanja, svako stanje računamo u $O(1)$ pa je konačna vremenska složenost ovog algoritma $O(n^3A)$.

4 Zadaci za samostalni rad

Neki lep zadatak sa takmičenja. Može i zadaci iz zbirke.

5 Zaključak

Savladali smo dinamičko programiranje.

Reference

[1] Bein W, Golin M.J, Larmore L.L, Zhang Y. *The Knuth-Yao quadrangle-inequality speedup is a consequence of total monotonicity.* TALG, vol. 6 (2009)

[2] Knuth D.E. *Optimum binary search trees*, Acta Informatica, vol. 1 (1971)

[3] Laaksonen A. *Guide to Competitive Programming*, Springer International Publishing (2017)

[4] Ambainis A, Filmus Y, Le Gall F. *Fast Matrix Multiplication: Limitations of the Coppersmith-Winograd Method.* STOC (2015)