Oblikovanje i analiza algoritama

Matej Mihelčić

Prirodoslovno-matematički fakultet, Sveučilište u Zagrebu

matmih@math.hr

03. studenoga, 2023.

- Dinamičko programiranje može **drastično skratiti** pretraživanje svih nizova odluka tako da **ne generira** neke od nizova koji sigurno nisu optimalni.
- Do optimalnog niza odluka dolazi se stalnim korištenjem **tzv. principa optimalnosti.**

Princip optimalnosti, R. Bellman, ~ 1957:

Optimalni niz odluka ima svojstvo da za bilo koje početno stanje i početnu odluku (u tom stanju), preostale odluke moraju činiti optimalan niz gledano iz stanja nakon prve odluke.

Napomena: ako je prva odluka bila pogrešna, optimalnost nadalje, neće dovesti do globalnog optimalnog rješenja. Još uvijek ostaje pitanje optimizacije prve odluke, ali samo **po optimalnim nizovima iza** te odluke.

Princip optimalnosti

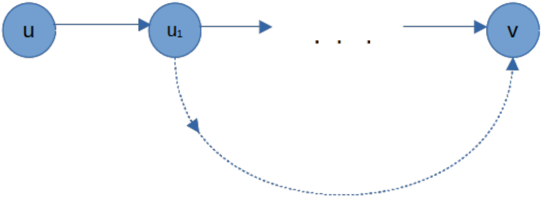
Zaključak:

- Primjena DP-a može generirati više nizova odluka za razliku od pohlepnog pristupa kojim se generira točno jedan niz.
- Globalni nizovi koji imaju neoptimalne podnizove sigurno nisu optimalni.

Da bismo neki problem riješili primjenom dinamičkog programiranja, trebamo:

- pronaći neki princip optimalnosti koji odgovara problemu
- dokazati da on vrijedi za taj problem ("Očigledno je." nije dokaz - na ovom kolegiju ☺).

Princip optimalnosti - problem najkraćeg puta



- Pretpostavimo da je $u, u_1, \ldots, v$ jedan optimalan put od u do v.
- Početna odluka u, u_1
- Tvrdimo da $u_1, \ldots, v$ mora biti optimalan put od u_1 do v.
- Ukoliko postoji kraći put od u_1 do v, tada postoji kraći put i od u do v.
- Princip optimalnosti **vrijedi** za problem najkraćeg puta i formuliran je **preko duljine najkraćeg puta**.

Problem najkraćeg puta

Nakon dokaza principa optimalnosti, algoritam slijedi sljedeće korake:

- ispitujemo početnu odluku za svaki grad u_1 direktno povezan s gradom u.
- od izabranog u_1 nadalje, tražimo najkraći put do v.
- rješenje je ukupno najkraći put (preko svih izbora u_1) koji od u prolazi do u_1 i od njega do v.

$l(u, v)$ - duljina najkraćeg puta između dva grada u i v.
$d(u, u_1)$ - duljina brida (ukoliko postoji) između gradova u i u_1.

$$l(u, v) = \min_{u_1 | \ (u,u_1) \in E} \{d(u, u_1) + l(u_1, v)\}$$

- nalaženje najkraćeg puta od u_1 do v **ne ovisi** o odluci u prvom koraku (korektan problem kojeg možemo riješiti bez poznavanja prve odluke)
- **općenito:** pronalazak preostalog optimalnog niza odluka ne ovisi o prvoj odluci (tzv. **princip neovisnosti ili invarijantnosti**).

Problem najkraćeg puta

- Problem nalaženja najkraćeg puta od u_1 do v je problem iste vrste kao i polazni problem.
- **općenito:** nalaženje preostalog optimalnog niza odluka je problem iste vrste kao i polazni (u široj klasi problema - parametrizacija). To je **princip ulaganja** u širu klasu problema ili **princip parametrizacije**.
- Za potproblem smijemo rekurzivno iskoristiti isti algoritam.
- Rekurzivna jednadžba po koracima (uz oznaku $u_0 = u$) :
$$l(u_k, v) = \min_{u_{k+1} |\ (u_k, u_{k+1}) \in E} \{d(u_k, u_{k+1}) + l(u_{k+1}, v)\},\ k = 0, 1, \dots$$
- Kraj puta su vrhovi direktno povezani s v - za koje $d(u_s, v) = l(u_s, v)$.
- Programska realizacija **ne mora biti rekurzivna**.

Primjeri algoritama:

- Dijsktrin algoritam: $\mathcal{O}(|V|^2)$ (original), $\Theta(|E| + |V| log_2 |V|)$ (fibonaccijeva hrpa). Najkraći put od jednog (zadanog) čvora - **izvora** do ostalih čvorova u grafu. Radi na usmjerenim težinskim grafovima s **nenegativnim** težinama.

Problem najkraćeg puta

- Bellman-Ford algoritam: $\Theta(|V||E|)$ u najgorem slučaju. Najkraći put od jednog (zadanog) čvora - **izvora** do ostalih čvorova u grafu. Radi na usmjerenim težinskim grafovima s **proizvoljnim** težinama.
- Floyd-Warshall algoritam: $\Theta(|V|^3)$. Najkraći put između svih parova čvorova u usmjerenom težinskom grafu s **proizvoljnim** težinama.
- Johnsonov algoritam: $\mathcal{O}(|V|^2 log_2|V| + |V||E|)$. Najkraći put između svih parova čvorova u usmjerenom težinskom grafu. Dopuštene negativne težine, međutim ne smiju postojati **ciklusi s negativnim težinama**.
- A*: $\mathcal{O}(|E| log_2|V|)$. Najkraći put od jednog (zadanog) čvora - **izvora** do jednog (zadanog) čvora - **odredišta** u usmjerenom težinskom grafu. Koristi heurističku funkciju h (ovisnu o problemu) za procjenu udaljenosti od trenutnog čvora do odredišta.

Princip optimalnosti - problem $0-1$ ruksaka

- Princip optimalnosti se razlikuje od problema do problema.
- Označimo s `KNAP(1,n,M)` problem za izbor predmeta s indeksima od l do n uz kapacitet M.
- Odluke su ili da ne stavimo neki predmet x_i u ruksak ($x_i = 0$) ili ga stavljamo u ruksak ($x_i = 1$).
- Ukoliko predmet x_1 stavimo u ruksak, trebamo puniti ruksak predmetima $2, \ldots, n$ uz dostupan kapacitet $M - w_1$.

Princip optimalnosti:

Neka je $y_1, \ldots, y_n$ optimalni niz 0 ili 1 vrijednosti za $x_1, \ldots, x_n$ u polaznom problemu `KNAP(1,n,M)`.

- Ako je $y_1 = 0$, onda podniz $y_2, \ldots, y_n$ mora biti optimalno rješenje problema `KNAP(2,n,M)`.
- Ako je $y_1 = 1$, onda podniz $y_2, \ldots, y_n$ mora biti optimalno rješenje problema `KNAP(2,n,M-`w_1`)`.

- Rješavanje problema KNAP(2,n,M) ili KNAP(2,n,M-w_1), ne ovisi o izabiru u prvom koraku. Vrijedi **princip invarijantnosti**.
- Pronalazk preostalog optimalnog niza odluka je problem iste vrste kao i polazni (u široj klasi problema), KNAP(2,n,M), KNAP(2,n,M-w_1), KNAP(3,n,M), ... Vrijedi **princip ulaganja**.

Rekurzivna jednadžba: neka je $g_j(M)$ vrijednost maksimalnog profita u optimalnom rješenju problema KNAP(1,n,M). Kao rješenje tražimo $g_1(M)$. Moguće odluke na početku su:

- $x_1 = 0$ pa je vrijednost profita $g_2(M)$
- $x_1 = 1$ pa je vrijednost profita $g_2(M - w_1) + p_1$

$g_1(M) = max\{g_2(M), g_2(M - w_1) + p_1\}$ (izabir $x_1 = 0$ ili $x_1 = 1$ radimo prema tome što je veće)

Rekurzivna primjena:

$g_i(M_i) = max\{g_{i+1}(M_i), g_{i+1}(M_i - w_i) + p_i\},\ i = 1, \dots, n$

Uzimamo: $g_{n+1}(M_i) = 0, \forall M_i$

Problem 0 – 1 ruksaka - implementacija

C implementacija (bottom - up):

```c
int max(int a, int b) { return (a > b) ? a : b; }

int knapsack(int M, int w[], int p[], int n){
    int i, wc;
    int K[n+1][M+1];

    for (i = n; i >= 0; i--) {
        for (wc = 0; wc <= M; wc++) {
            if (i == n || wc == 0)
                K[i][wc] = 0;
            else if(w[i]<=wc)
                K[i][wc] = max(p[i] +
                               K[i+1][wc - w[i]],
                               K[i + 1][wc]);
            else
                 K[i][wc] = K[i + 1][wc];
        }
    }
    return K[0][M]; }
```

C implementacija (top - down):

```c
int knapsacktd(int M, int w[], int p[], int index, int** K,
    int maxN){
   if (index >= maxN)
       return 0;
   if (K[index][M] != -1)
       return K[index][M];

   if (w[index] > M) {
       K[index][M] = knapsacktd(M, w, p, index + 1, K,maxN);
       return K[index][M];}
    else {
        K[index][M] = max(p[index] +
                        knapsacktd(M - w[index], w, p,
                                   index + 1, K,maxN),
                     knapsacktd(M, w, p, index + 1, K,maxN));

        return K[index][M];
    }
}
```

Problem 0 – 1 ruksaka - implementacija

```c
int knapsack(int M, int w[], int p[], int n)
{
    int **K;
    K = (int**) malloc(n*sizeof(int*));

    for(int i=0;i<n;i++)
        K[i] = (int*) malloc ((M+1)*sizeof(int));

    for (int i = 0; i < n; i++)
        for (int j = 0; j < M + 1; j++)
            K[i][j] = -1;

    return knapsacktd(M, w, p, 0, K,n);
}
```

Primjer

Interpretirajte jednadžbu:

$$a(i,j) = \begin{cases} 0, & i > j \\ 1, & i = j \\ a(i+1, j-1) + 2, & x[i] = x[j] \\ max\{a(i+1,j), a(i,j-1)\}, & x[i] \neq x[j] \end{cases}$$

Opišite rekurzivnu implementaciju.

- Pohlepni (greedy) algoritmi:
  - najčešće jednostavni
  - koriste se (uglavnom) za rješavanje problema optimizacije
    - minimalno razapinjuće stablo
    - najbolji redoslijed izvođenja nekih poslova na računalu
- Karakteristični elementi:
  - C - skup svih raspoloživih kandidata (bridovi, poslovi, ...)
  - S - skup izabranih kandidata
  - funkcija izbora (selekcije) koja u svakom trenutku odabire najperspektivnijeg, još neizabranog kandidata $x \in C$.
  - funkcija koja provjerava je li izabrani skup kandidata dopustiv (feasible) u smislu da dodavanjem izabranog elementa dobivamo bar jedno (pod)rješenje, ne nužno optimalno.
  - funkcija cilja (objective function) koju optimiziramo (duljina razapinjućeg stabla, vrijeme potrebno za izvođenje poslova u danom nizu i sl.).

Pohlepni algoritmi

Pohlepni algoritam **napreduje korak po korak**:

- inicijalizacija: $S \leftarrow \emptyset$
- sve dok nismo pronašli rješenje i $C \neq \emptyset$
  - odabrati $x \in C$ koji optimizira vrijednost funkcije cilja
  - $C \leftarrow C \setminus \{x\}$
  - ako je $S \cup \{x\}$ dopustiv tada $S \leftarrow S \cup \{x\}$

Svojstva:

- Pohlepni algoritam:
  - u svakom koraku bira najveći zalogaj koji može progutati (pohlepa), ne vodeći računa o budućnosti ili o prošlosti
  - nikada ne mijenja odluku – kada je kandidat jednom uključen u rješenje on tamo i ostaje, odnosno kada je jednom odbačen više ga ne provjeravamo
  - ne mora naći rješenje
  - nađeno rješenje ne mora biti optimalno.
- Pohlepni algoritmi su brzi (svaki kandidat se provjerava najviše jednom).

Pohlepni algoritmi

Svojstva:

- Napomena. Funkcija izbora je najčešće bazirana na funkciji cilja (mogu biti identične)

Primjena:

- rješavanje teških problema optimizacije kao brza heuristika koja daje neko rješenje iako ne optimalno (TSP – Travelling Salesman Problem)
- pronalaženje početnog rješenja iterativnih metoda optimizacije koje poboljšavaju postojeća rješenja.