DINAMIČKO PROGRAMIRANJE

Dinamičko programiranje (DP) je jedna od metoda za konstrukciju algoritama.

Još neke od metoda za konstrukciju algoritama su:
- pohlepna (greedy) metoda
- podijeli, pa vladaj (divide and conquer)
- backtracking
- branch and bound.

⋮

U ovom kontekstu, mislimo prvenstveno na tzv. kombinatorne algoritme - probleme prebrajanja, probleme na grafovima i sličnim diskretnim strukturama. Posebnu, veliku i važnu klasu čine problemi kombinatorne optimizacije, gdje tražimo rješenja koja su optimalna u nekom smislu (najkraći put, najmanja cijena, maksimalni profit, ...).

DP se može primijeniti na probleme čije rješenje možemo interpretirati kao rezultat nekog niza odluka.

Kao primjer, uzmimo slijedeća dva problema:

Problem 1. [Najkraći put]

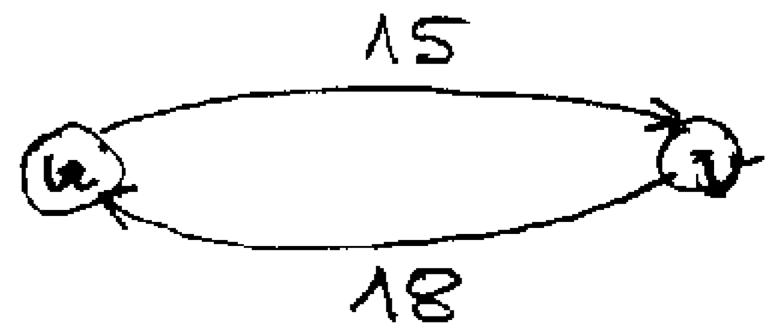
Zadano je n gradova - označimo ih brojevima od 1 do n. Između pojedinih gradova postoje direktne veze, ali dozvoljavamo da neki gradovi nisu direktno vezani (bez prolaska kroz druge gradove).

Tako dobivamo jedan graf G, u kojem su čvorovi ili vrhovi - skup V - gradovi, a bridovi ili grane - skup E - direktne veze između gradova.

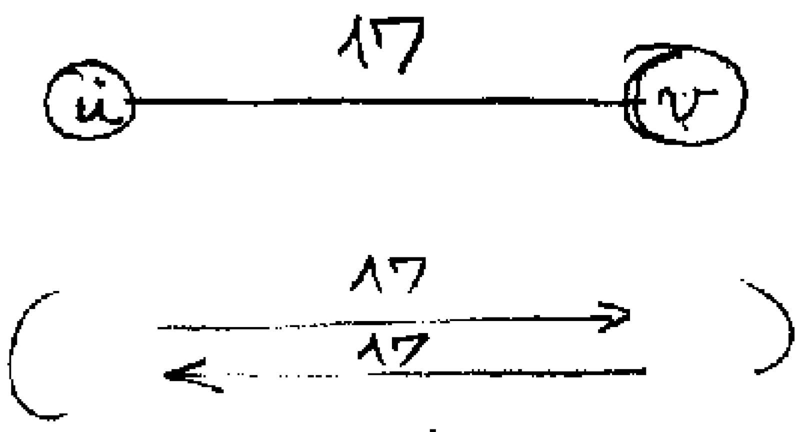
Općenito dozvoljavamo jednosmjerne veze, tj. kažemo da veza ide iz grada "u" u grad "v" - kao uređeni par (u, v).

Svaka takva direktna veza ima neku zadanu duljinu $\ell(u,v)$ ili, općenito neku cijenu (ili trajanje)

Uočimo da jednosmjerne veze dozvoljavaju da postoji i veza obratnog smjera, ali da njena duljina (cijena) ne mora biti ista kao za vezu prvotnog smjera. Takav graf zovemo <u>usmjereni graf</u>. Ako su sve veze dvosmjerne i oba smjera imaju uvijek istu duljinu dobivamo <u>neusmjereni graf</u> – bridovi su tada dvočlani skupovi vrhova – oblika $\{u, v\}$.



(na pr. u je na planini, a v u dolini, pa je spuštanje jeftinije ili brže od penjanja)



pa smjer ne pišemo

Pošto ne mora svaki grad biti direktno vezan sa svakim, očito se postavljaju dva problema:

- za zadane gradove u, v, da li postoji neki put (kroz neki niz gradova) kojim se može stići iz u u v?

  To je tzv. problem povezanosti grafa, koji se može efikasno riješiti.
  Mi ćemo pretpostavljati da je naš graf povezan, tj. da možemo iz bilo kog grada stići u neki drugi grad.

- naći najkraći put od u do v. To znači naći niz gradova $u_1, u_2, \dots, u_k$ koji su direktno vezani

$$u \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \dots \longrightarrow u_k \longrightarrow v$$

između u, v, tako da je ukupna duljina ovog puta najkraća moguća.
(duljina puta je, naravno, <u>zbroj</u> duljina svih pojedinih grana)

Između ta dva grada može postojati mnogo raznih puteva. Nalaženje rješenja možemo interpretirati kao niz odluka:

- u koji grad u_1 treba krenuti iz grada u,
- u koji grad u_2 ——— " ——— u_1,

⋮

Uočimo odmah da odluke nisu nezavisne!

Problem 2. [Ruksak]

Imamo na raspolaganju ruksak kapaciteta M (recimo da je to najveća težina koju možemo ponijeti. Druga varijanta – to je volumen ruksaka). Taj ruksak treba napuniti objektima. Možemo birati između n vrsta objekata. Jedinična količina objekta vrste i ima težinu w_i (ili volumen!) u ruksaku i donosi nam profit p_i ako ju ponesemo.

Treba izabrati količine x_i objekata pojedinih vrsta ($i = 1, \dots, n$) koje ćemo staviti u ruksak i to tako da ne prepunimo ruksak, a da ukupni profit bude maksimalan.

Količina x_i predmeta i-te vrste ima težinu $w_i x_i$ i profit $p_i x_i$.

Dakle, treba naći $x_1, \dots, x_n$ tako da bude

$$\text{uk\_profit} = \sum_{i=1}^{n} p_i x_i \to \max$$

$$\text{uk\_težina} = \sum_{i=1}^{n} w_i x_i \leq M.$$

– Ovaj problem je lako rješiv, ako imamo neograničene količine objekata na raspolaganju. Treba, naravno, gledati koja vrsta objekata daje najveći profit po količini, tj. koja daje najveći omjer

$$\frac{p_i}{w_i}$$

i tom vrstom potpuno ispuniti ruksak ($x_i = \frac{M}{w_i}$ za taj i, a sve ostale količine su $\emptyset$).

- U praksi, količine nisu neograničene, već imamo zadane maksimalne količine X_i objekata i-te vrste. Dakle, dodatni zahtjev je

$$x_i \leq X_i \quad , \text{ za } i = 1, \ldots, n.$$

I to je lako. Ruksak punimo redom najvrednijim objektima (silazno po p_i/w_i) i to maksimalnim količinama, dok ide. Na kraju dopunimo ruksak do kapaciteta, najvrednijim preostalim objektom.

- Ako su vrste predmeta tipa brašno, šećer, kava, ulje, čaj i sl. onda x_i može biti bilo kakva količina (realan broj).
- Međutim, planinar ne može ponijeti 1/2 cuture ili $\emptyset$.26 četkica za zube.

  Vrlo često ograničenje je uvjet da x_i još mora biti i (nenег.) cijeli broj - tj. predmete ne smijemo rezati.

  Tako dobivamo tzv. cjelobrojni ruksak (Integer Knapsack) - problem koji je mnogo teži.
- Standardna varijanta ovog problema je tzv. $\emptyset$-1 ruksak [$\emptyset$-1 Knapsack] u kojem su količine x_i ograničene na vrijednosti $\emptyset$ ili 1

$$x_i \in \{\emptyset, 1\} \quad , i = 1, \ldots, n.$$

(Što će vam 12 četkica za zube).

Tj. predmet i-te vrste uzimamo ili ne uzimamo!

- Promatrat ćemo ovaj oblik problema, koji je i dalje težak, u smislu da ne postoji efikasni algoritam za njegovo rješavanje - čije trajanje bi bilo polinomno u broju predmeta n (n^2, n^3 ili bar n^{1994})

Odluke pri rješavanju ovog problema su vrlo očite:

- izaberi količinu x_1,
- izaberi količinu x_2,

  ⋮

- izaberi količinu x_n.

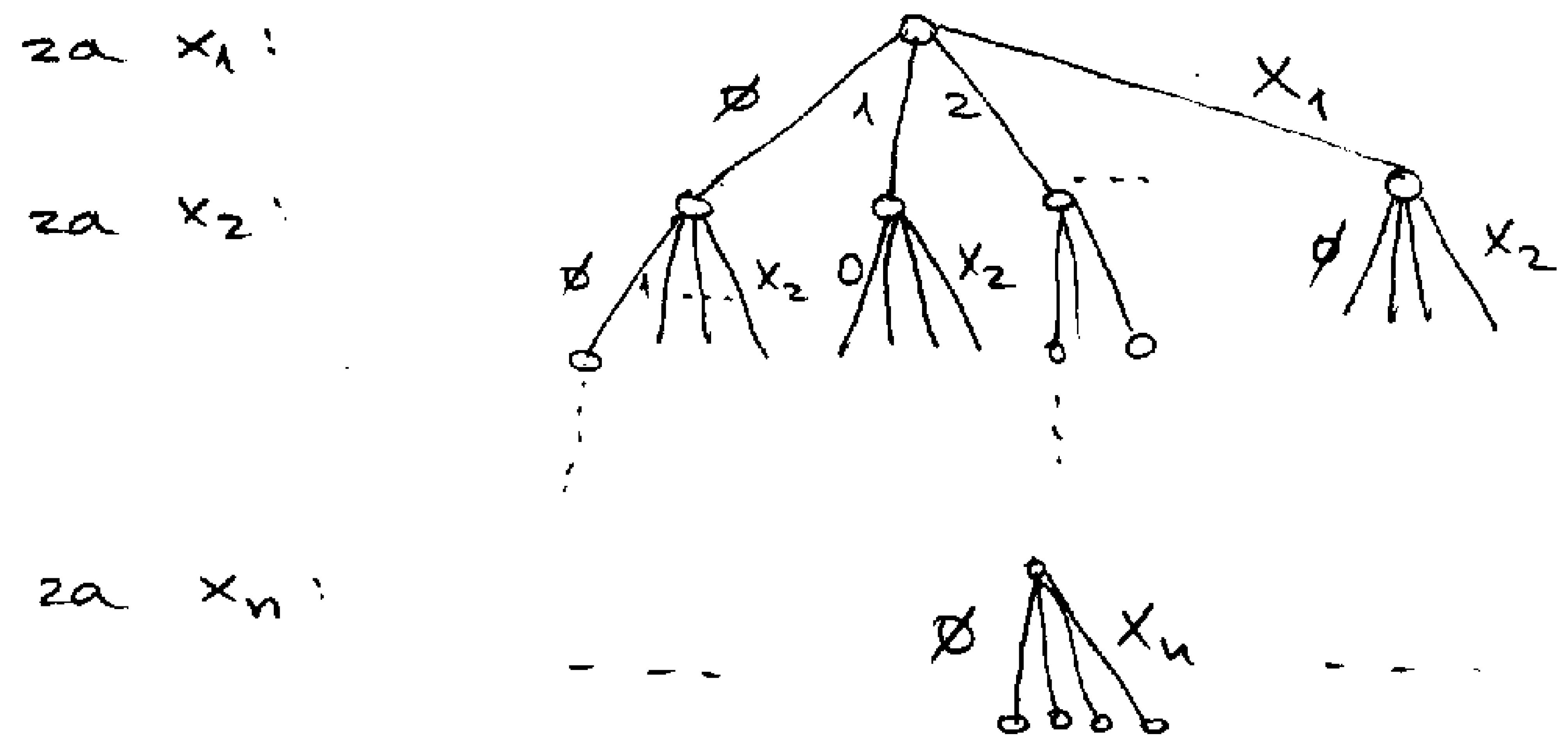
Za $\emptyset$-1 ruksak – još je očitije:

- odluči: ponijeti x_1 ili ne,

  ⋮

- odluči: – " – x_n ili ne.

- Taj postupak odlučivanja – bolje rečeno, mogućnosti raznih odluka možemo lijepo prikazati stablom:



Broj mogućnosti je ogroman: $(X_1+1)\cdot(X_2+1)\cdots(X_n+1)$ za cjelobrojni ruksak

odnosno $\underline{2^n \text{ za } \emptyset\text{-1 ruksak}}$.

- Ako pišemo glupi program, on će morati <u>pretražiti</u> svih 2^n mogućnosti u ovom stablu odluka.
- Osnovna ideja je da pametnim odlučivanjem smanjimo broj mogućnosti koje treba pretražiti. Većina tehnika za konstrukciju algoritama (greedy, DP, backtrack, branch-and-bound) ima upravo taj cilj → <u>što brže naći optimalan niz odluka</u>.

Za neke probleme ovog tipa, optimalni niz odluka može se naći tako da u svakom koraku donosimo najbolju odluku (jednu po jednu), a da time nikad nismo pogriješili.

Upravo to je pohlepna ili greedy metoda.

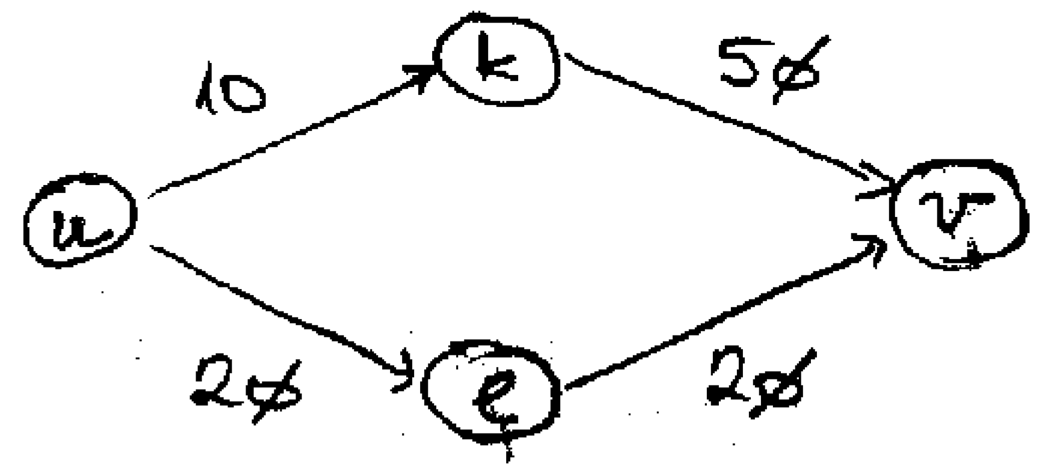
Za mnoge druge probleme, nije moguće u pojedinim koracima donositi odluke, na osnovu samo lokalnih informacija, tako da čitav niz odluka bude optimalan.

Pokažimo to na primjerima u našim problemima.

Primjer 1. [Najkraći put]

Tražimo najkraći put iz grada u u grad v. Neka je A_u skup svih gradova u koje možemo doći direktno iz grada u. Koji od gradova iz A_u treba biti slijedeći grad na putu u v?

Nema načina da to lokalno odlučimo (bez gledanja ostatka grafa) ovog trena, a da garantiramo da buduće odluke vode na optimalan niz.



k je bliži u od l, ali je ipak bolje ići kroz l.

Međutim, ako proširimo problem i tražimo najkraće puteve od u do svih ostalih gradova (u koje možemo doći), jasno je da možemo donijeti korektnu odluku i to pohlepno!

Time dobivamo tzv. Dijkstrin algoritam za sve najkraće puteve iz jednog vrha

$u \to k$ 10
$u \to l$ 20

i to su jedini koje možemo direktno. Sad gledamo sve gradove u koje možemo direktno iz ovih. To je samo v – i biramo kraće varijante (opet pohlepno po ukupnoj duljini)

$u \to k \to v$ 60
$u \to l \to v$ 40 $\Rightarrow$ $u \to v$ 40.

Primjer 2. [0-1 ruksak]

Uzeti prvi predmet ili ne? Kako god odlučili možemo i pogriješiti – dok ne vidimo što je s ostalim predmetima:

i	w_i	p_i
1	4	3
2	3	2
3	2	2

(a) $M = 5$ i uzmemo prvog $x_1 = 1$ – više ništa ne stane ($w_1 = 4$), pa je ukupni profit samo 3. Bolje je uzeti drugi i treći pa je $x_2 = x_3 = 1$, a profit je $p_2 + p_3 = 4$.

(b) $M = 4$ i ne uzmemo prvog, opet smo pogriješili.

Naprotiv, ako su $x_i \in \mathbb{R}$, tj. predmete smijemo rezati, pohlepni algoritam stavljanja najvrjednijih predmeta uredno daje rješenje (nakon sortiranja po p_i / w_i).

— · —

Dinamičko programiranje može drastično skratiti pretraživanje svih nizova odluka tako da preskače (ne generira) neke od nizova koji sigurno nisu optimalni.

Do optimalnog niza odluka (ili svih takvih, ako ih je više), dolazi se stalnim direktnim korištenjem tzv. principa optimalnosti, ako nađemo neki takav princip za naš problem (što treba dokazati).

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 ne po svim, nego samo optimalnim nizovima iza te odluke!

Iz ovoga možemo zaključiti:

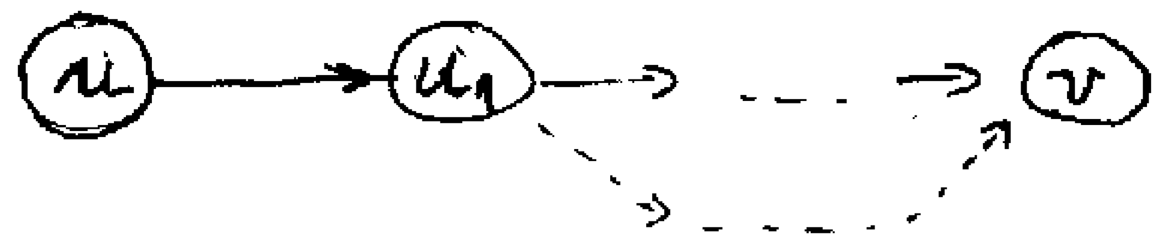
- Primjena DP može generirati mnogo nizova odluka, za razliku od pohlepe, koja generira točno jedan niz.
- Međutim, globalni nizovi koji imaju neoptimalne podnizove [na krajnjem svom dijelu] sigurno nisu optimalni (ako vrijedi princip optimalnosti) i neće biti niti generirani (u najvećoj mogućoj mjeri).

Da bismo neki problem riješili primjenom DP treba:

- pronaći neki princip optimalnosti (koji odgovara problemu),
- dokazati da on vrijedi za taj problem (što je obično relativno očito, ali treba biti oprezan s očitim stvarima).

<u>Primjer</u> [Najkraći put]

Neka je $u, u_1, u_2, \dots, u_k, v$ najkraći put od u do v. U polaznom gradu u, donijeli smo odluku da odemo u grad u_1. Nakon te odluke, nalazimo se u gradu u_1 i trebamo pronaći najkraći put do v. Imamo opet problem najkraćeg puta. Očito je da niz $u_1, u_2, \dots, u_k, v$ mora biti i najkraći put od u_1 do v. U protivnom, da ima kraći put od u_1 do v, dobili bismo i kraći put od u do v.



Dakle, princip optimalnosti vrijedi za ovaj problem (i formuliran je preko duljine najkraćeg puta - to treba reći, iako je to ovdje očito).

Uočimo: DP odbacuje sve neoptimalne puteve od u_1 do v.
(odn. princip opt.)

Nakon što smo ovo uočili (i dokazali) naš algoritam za traženje najkraćeg puta od u do v izgledao bi ovako:

- gradova u_1 može biti puno i sve te početne odluke treba provjeriti, jer ne znamo koja je optimalna;
- od bilo kog u_1 nadalje, naći najkraći put do v.
- uzmi najkraći ukupni put po svim u_1.

ld - least distance

Označimo s $ld(u,v)$ duljinu optimalnog (najkraćeg) puta od u do v, za bilo koja 2 grada.

Onda je:

$$ld(u,v) = \min_{\substack{\text{po svim } u_1 \\ \text{takvim da je } (u,u_1)\in E}} \{ l(u,u_1) + ld(u_1,v) \} .$$

Da bismo iz ovog konstruirali algoritam za rješenje čitavog problema, treba uočiti još dvije stvari:

- nalaženje najkraćeg puta od u_1 do v <u>ne</u> ovisi o odluci u prvom koraku (to je korektan problem kojeg možemo samostalno riješiti, bez da znamo da smo u u_1 stigli iz u)

  <u>općenito</u>: preostale odluke ili preostali optimalni niz odluka (za potproblem na repu) ne ovise o prvoj odluci
  — tzv. <u>princip neovisnosti</u> ili invarijantnosti;

- zbog toga je problem nalaženja najkraćeg puta od u_1 do v, opet problem istog tipa kao i polazni problem, samo na neka druga dva grada (prvi grad varira!).

  <u>općenito</u>: nalaženje preostalog optimalnog ~~niza odluka~~ je problem iste vrste kao i polazni, samo u široj klasi problema (parametrizacija problema) - tzv. <u>princip ulaganja</u> u širu klasu problema ili <u>princip parametrizacije</u>.

široj = ne samo od u do v, već i za u_1, ..., Zbog toga smijemo i za potproblem rekurzivno primijeniti isti algoritam.

Uočiti – i invarijantnost je bitna, iako izgleda sasvim očito, ali nije uvijek.
(ako ovisimo o ranijim odlukama dobivamo backtracking algoritme!)

– programska realizacija ne mora biti rekurzivna, ono što je bitno to je <u>rekurzivna jednadžba</u> po koracima:

Uz oznaku $u_0 = u$, u bilo kojem koraku vrijedi:

$$ld(u_k, v) = \min_{\text{po svim } u_{k+1}} \left\{ l(u_k, u_{k+1}) + ld(u_{k+1}, v) \right\}$$

za dani u_k, gledamo sve u_{k+1} u koje možemo doći iz u_k, tj. $(u_k, u_{k+1}) \in E$.

$k = 0, 1, \ldots$

Da bismo ovo započeli rješavati, rekurzija mora imati kraj. Ovdje, na općenitom grafu, kraj nije očit, ali jasno je da rješavanje treba provoditi <u>unatrag</u> – od v (u obratnom smjeru ništa ne dobivamo – prolazimo cijelo stablo odluka).

Na kraju puteva su oni čvorovi koji su <u>direktno</u> vezani s v, tj za koje je

$$ld(u_{k+1}, v) = l(u_{k+1}, v).$$

Takvi se lako nalaze, jer to je zadani podatak. Sad kad smo počeli, možemo ići unatrag. Koraka ima najviše $n-1$ (jer imamo n gradova), a "šetajući silazno po k", stajemo kad unazad naletimo na $u_k = u$.

- Treba priznati da indeks k ovdje samo smeta, jer ne možemo reći da on prolazi nekim fiksnim skupom vrijednosti. Zbog toga je direktna programska realizacija malo nezgodna – jer razni putevi od u do v mogu imati razni broj gradova (vrhova) na tom putu.

Također, treba oprezno realizirati algoritam, prateći već prođene vrhove na putu. U protivnom, možemo naići na cikluse. Negativni ciklusi, očito, nisu dozvoljeni, ili treba pojačati zahtjev – da nema ponavljanja gradova na putu (što, opet, treba pamtiti).

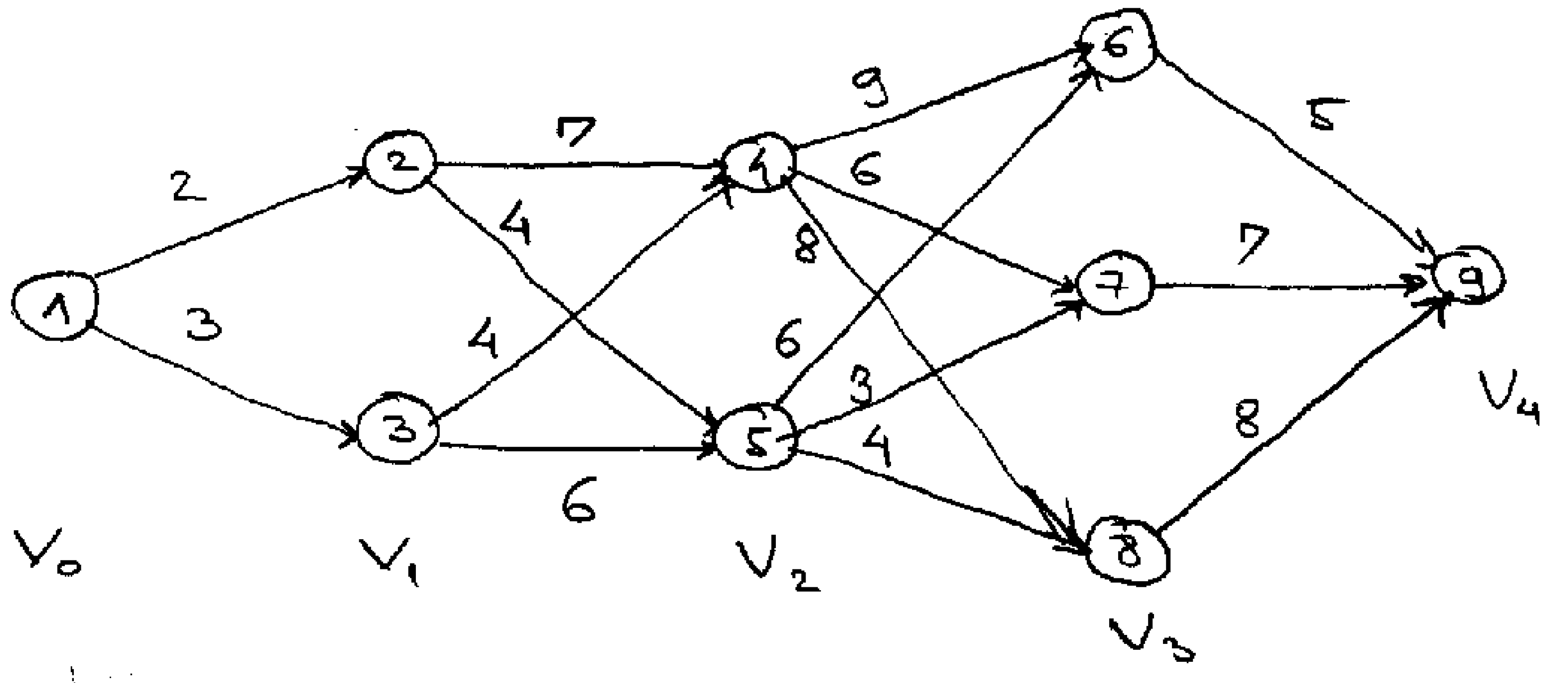
- Treba priznati da ovdje indeks k samo smeta, jer ne možemo reći da on prolazi fiksnim skupom vrijednosti. Zbog toga je programska realizacija malo nezgodna, jer razni putevi od i do j mogu imati razni broj gradova između ta dva grada.
- Jednostavno se realiziraju 2 varijante ovog problema.

(A) Za rješenje ionako koristimo isti algoritam s različitim početnim gradovima, pa možemo naći rješenje za <u>sve</u> gradove u, uz fiksni v. (inverzni Dijkstrin alg.).

Još je lakše realizirati algoritam koji nalazi $ld(u,v)$ za <u>sve parove</u> (u,v), $u \neq v$.

To je tzv. Floyd-ov algoritam (Sec. 5.4, str. 151 u Algorithmics)

(B) Zadani usmjereni graf je <u>slojevit</u>. U takvom grafu, vrhovi su podijeljeni u $m+1$ disjunktnih skupova (slojeva), $m \geq 1$. Označimo te skupove s $V_0, V_1, \ldots, V_m$. Direktne veze postoje samo između vrhova u susjednim slojevima V_k i V_{k+1} (ne mora svaki par biti vezan!)



Ovdje su svi parovi vrhova u susj. slojevima vezani.

Čvorovi iz sloja V_0 zovu se izvori, a oni iz zadnjeg sloja V_m ponori.

- Problem najkraćeg puta u slojevitom grafu je:
  Za zadani izvor $u \in V_0$ i ponor $v \in V_m$, naći $ld(u,v)$.

Svaki put od izvora do ponora ima točno m dionica (komada) – na svakom sloju po jedan grad ($m+1$ vrh, zajedno s u, v).

Ranija rekurzivna jednadžba postaje:

$$ld(u_k, v) = \min_{\substack{\text{po svim } u_{k+1} \in V_{k+1} \\ \text{s kojima je } u_k \in V_k \\ \text{vezan}}} \{ l(u_k, u_{k+1}) + ld(u_{k+1}, v) \}$$

za $k = 0, 1, \ldots, m-2$ (može i $m-1$, uz dogovor $ld(u_m, v) = 0$) $[u_m \in V_m \Rightarrow u_m = v]$

i start na zadnja 2 sloja:

$$ld(u_{m-1}, v) = l(u_{m-1}, v)$$

za sve u_{m-1} iz kojih se može u v.

- Zadatak: Naći $l(1,9)$ za graf sa slike i pripadni najkraći put.

Rješenje: $1 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 9$, $l(1,9) = 16$.

- <u>Algoritam</u>: Uočimo prvo da koristimo samo po jedan vrh iz prvog i zadnjeg sloja ($u \in V_0$, $v \in V_m$)
Možemo stoga pretpostaviti da su V_0, V_m jednočlani.
Nadalje, vrhove (gradove) numeriramo redom po slojevima brojevima od 1 do n:
izvor → broj 1, vrhovi iz V_1, pa $V_2, \ldots$, ponor → broj n
i zapravo tražimo $ld(1, n)$.

- Korisno je da vrhovi iz V_{k+1} imaju veće brojeve od onih iz V_k.
- Također, korisno je pamtiti indekse u_{k+1} na kojima se dostižu minimumi u rekurziji, jer upravo ti gradovi ulaze u optimalni put od 1 do n.

```
procedure SPATH ( E, m, n, P, l );
 { Ulaz je graf s m+1 slojeva i n vrhova,
   indeksiranih rastuće po slojevima.
   E je skup bridova - direktnih veza, a d(i,j)
   je duljina brida (i,j), i < j.
Izlaz: Polje P(1:m) sadrži redom indekse gradova
   na najkraćem putu  P(k) = i_k, k = 1,..,m
   l - duljina najkraćeg puta }

var   ld (1:n) - polje koje pamti najkraće duljine
                 ld(j) = l(j,n);
      ind (1:n-1) - pamti indekse za koje se postižu
                    minimumi

begin
   ld(n) ← ∅;   { l(n,n) = 0 }
   for j ← n-1 downto 1 do
      begin
       neka je  r  vrh takav da je  (j,r) ∈ E, tj.
         j i r su direktno vezani i
              d(j,r) + ld(r)  je najmanje
         za sve takve r;
       ld(j) ← d(j,r) + ld(r);  {taj najkr. put
                                  l(j,n)}
       ind (j) ← r ;  { min. indeks }
      end;
   P(1) ← ind(1); { start puta }
   for j ← 2 to m-1 do
      P(j) ← ind (P(j-1)) ;
   P(m) ← n;
end;  {SP}
```

```
procedure SP_Layered ( n, m : integer;
                       E : bridovi; {s udaljenostima}
                       var P : polje_vrhova {indeksi 1..m};
                       var d : real );

{ Ulaz: slojeviti graf G=(V,E) s n vrhova, V={1,...,n},
        numeriranih rastuće po slojevima od V0 do Vm.
        Broj slojeva je m+1.
        E je skup bridova, a struktura bridovi sadrži
        i udaljenosti L(i,j), za brid (i,j) ∈ E, i < j.
  Izlaz: Polje P[1..m] sadrži, redom, indekse (brojeve)
        vrhova na najkraćem putu od 1 do n,
         P[k] = u_k,  k=1,...,m.
        d = duljina najkraćeg puta od 1 do n }

{ Lokalna polja:
    LD [1..n] - pamti najmanje udaljenosti unatrag
                LD [j] = ld (j, n);
    ind [1..n-1] - pamti indekse (vrhove) za koje se
                   postižu minimumi. }

begin
  LD[n] := 0.0;   { l(n,n) = 0 - dogovor }
  for j := n-1 downto 1 do
    begin
    naoti vrh r, takav da je (j,r) ∈ E i vrijedi
          L(j,r) + LD[r] je najmanje, za sve takve r;
    LD[j] := L(j,r) + LD[r];
    ind[j] := r;   { iz j treba ići u r }
    end;
    { formiraj najkraći put - unaprijed }
  P[1] := ind[1];  { početni vrh u1 }
  for j := 2 to m-1 do
    P[j] := ind[P[j-1]];
  P[m] := n;
  d := LD[1];
end; { SP_Layered }
```

<u>Složenost</u>:

- najveći problem je nalaženje vrha r takvog da je $L(j,r) + LD[r]$ najmanje.

Ako graf prikazujemo listama susjedstva, treba samo pretražiti sve susjede vrha j. Trajanje je proporcionalno stupnju tog vrha.

Gledano po svim vrhovima - zbrojeno - ukupno trajanje je proporcionalno broju $e = |E|$ bridova.

Čitav proces se odvija u petlji, pa treba dodati $\Theta(n)$ vremena za tu petlju

Kraj troši $\Theta(m)$, a zbog $m < n$, to je već pokriveno s $\Theta(n)$.

Dakle, ukupno vrijeme je

$$T(n,e) = \Theta(n+e).$$

<u>Napomena</u>: Princip optimalnosti može se formulirati i "unatrag".

Označimo s A'_v sve gradove <u>iz</u> kojih možemo doći u traženi grad v

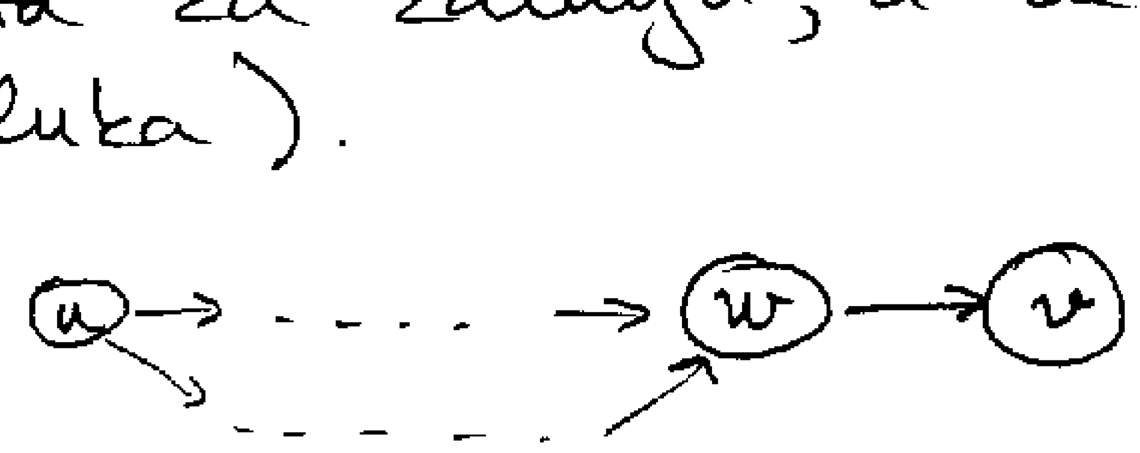
$$A'_v = \{ w \in V \mid (w,v) \in E \} \qquad w \longrightarrow v$$

Da bismo dobili najkraći put od u do v, treba promatrati najkraće puteve do vrhova iz A'_v i odabrati onog koji je najkraći, kad se doda put od w do v.

Tj:

$$ld(u,v) = \min_{\substack{\text{po svim } w \in A'_v \\ \text{tj. } (w,v) \in E}} \{ ld(u,w) + \ell(w,v) \}$$

Ovo je parametrizacija po <u>dolaznom vrhu</u> (v) i lako se vidi da vrijedi princip invarijantnosti (ovoga puta za zadnju, a ne za prvu odluku, u nizu odluka).



Rekurzivna jednadžba ima oblik

$$ld(u, u_{k+1}) = \min_{\substack{\text{po svim } u_k, \\ (u_k, u_{k+1}) \in E}} \{ ld(u, u_k) + l(u_k, u_{k+1}) \} \qquad k = 0, 1, \ldots$$

uz dogovor $u_0 = u$ i $ld(u, u_0) = l(u, u_0) = 0$, tj startamo sprijeda - iz u i gledamo vrhove direktno vezane s u:

$$ld(u, u_1) = l(u, u_1)$$
$$\forall u_1 \in V \text{ t.d. } (u, u_1) \in E.$$

Postupak rješavanja ide unaprijed, dok ne dobijemo $u_{k+1} = v$.

Princip optimalnosti i rekurzivne jednadžbe, razlikuju se od problema do problema:

<u>Primjer</u> [$\emptyset$-1 ruksak]

Prvo, kako parametrizirati? Odluke su redom $x_i = 0$ ili $x_i = 1$. Stavimo li $x_1 = 1$, ostaje napuniti ruksak predmetima od 2 do n, ali kapacitet pada na $M - w_1$.

Stoga označimo općenito s KNAP(ℓ, r, K) problem za izbor predmeta s indeksima od ℓ do r (pri varira!) uz kapacitet K.

<u>Princip optimalnosti</u>:

Neka je $y_1, \ldots, y_n$ optimalni niz $\emptyset$ 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 $y_2, \ldots, y_n$ mora biti optimalno rješenje za problem KNAP$(2, n, M - w_1)$.

<u>Invarijantnost i ulaganje</u>: je očito!

<u>Rekurzivna jednadžba</u>: Neka je $g_j(K)$ vrijednost maksimalnog profita u opt. rješ. za KNAP(j, n, K). Mi tražimo kao rješenje $g_1(M)$. Moguće odluke na početku su:

$x_1 = 0$, pa je vrij. profita $g_2(M)$

ili $x_1 = 1$, — " — $g_2(M - w_1) + p_1$

Dakle:

$$g_1(M) = \max\{ g_2(M), g_2(M - w_1) + p_1 \}$$

a iz toga što je veće, zaključujemo treba li uzeti $x_1 = 0$ ili $x_1 = 1$.

Rekurzivna primjena daje

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

za $i = 1, \ldots, n$.

Ovo vrijedi i za $i=n$, uz dogovor

$$g_{n+1}(K) = \emptyset, \quad \forall K$$

jer, bez obzira na kapacitet, nemamo što staviti u ruksak, pa je profit nula.

To omogućava silazno rekurzivno rješavanje, s tim da trebamo dodatnu memoriju za pamćenje rješenja ("indeksa" ili odluke, na kojem se dostiže max).

Pogled unatrag na niz odluka $x_1, \ldots, x_n$:

Neka je $f_j(K)$ vrij. maksimalnog profita u optimalnom rješenju problema KNAP$(1, j, K)$.

Tada je:

$$f_j(K) = \max\{\underbrace{f_{j-1}(K)}_{x_j=0},\ \underbrace{f_{j-1}(K-w_j) + p_j}_{x_j=1}\}. \quad j = 1, \ldots, n \text{ (v. dole)}$$

Traženo rješenje cijelog problema je $f_n(M)$. Možemo ga dobiti rješavanjem rekurzije unaprijed.

Start je:

$$f_0(K) = 0, \quad \forall K \geq 0$$

(ništa nismo stavili, a imamo pozitivni/neneg. kapacitet)

ali treba dodati i

$$f_0(K) = -\infty, \quad \forall K < 0$$

jer $K - w_j$ može biti negativan, ako w_j (j-ti objekt) više ne stane u ruksak. (Dovoljno je za $j=1$, ti w_1 ne stane).